O'REILLY®

TURING 图灵程序设计丛书



Python 网络数据采集

Web Scraping with Python

[美] Ryan Mitchell 著

陶俊杰 陈小莉 译

中国工信出版集团

人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

译者介绍

陶俊杰

长期从事数据分析工作，酷爱Python，每天都和Python面对面，乐此不疲。本科毕业于北京交通大学机电学院，硕士毕业于北京交通大学经管学院。曾就职于中国移动设计院，目前在京东任职。喜欢遛狗、看书、修电脑、翻译。

陈小莉

长期从事数据分析工作，喜欢Python。本科与硕士毕业于北京交通大学电信学院。目前在中科院从事科技文献与专利分析工作。喜欢遛狗、看书，自命为烹饪大师。

TURING 图灵程序设计丛书

Python网络数据采集

Web Scraping with Python

Collecting Data from the Modern Web

[美] Ryan Mitchell 著
陶俊杰　陈小莉 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　京

图书在版编目（C I P）数据

Python网络数据采集 / (美) 米切尔 (Mitchell,R.)
著 ; 陶俊杰, 陈小莉译. -- 北京 : 人民邮电出版社,
2016.3
（图灵程序设计丛书）
ISBN 978-7-115-41629-2

Ⅰ. ①P… Ⅱ. ①米… ②陶… ③陈… Ⅲ. ①软件工
具－程序设计 Ⅳ. ①TP311.56

中国版本图书馆CIP数据核字(2016)第023324号

内 容 提 要

本书采用简洁强大的 Python 语言，介绍了网络数据采集，并为采集新式网络中的各种数据类型提供了全面的指导。第一部分重点介绍网络数据采集的基本原理：如何用 Python 从网络服务器请求信息，如何对服务器的响应进行基本处理，以及如何以自动化手段与网站进行交互。第二部分介绍如何用网络爬虫测试网站，自动化处理，以及如何通过更多的方式接入网络。

本书适合需要采集 Web 数据的相关软件开发人员和研究人员阅读。

◆ 著 [美] Ryan Mitchell
译 陶俊杰 陈小莉
责任编辑 岳新欣
执行编辑 李 敏
责任印制 杨林杰

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京 印刷

◆ 开本：800×1000 1/16
印张：13.5
字数：280千字 2016年 3 月第 1 版
印数：1－4 000册 2016年 3 月北京第 1次印刷
著作权合同登记号 图字：01-2015-8108号

定价：59.00元

读者服务热线：(010)51095186转600 印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

第二部分　高级数据采集

译者序

每时每刻，搜索引擎和网站都在采集大量信息，非原创即采集。采集信息用的程序一般被称为网络爬虫（Web crawler）、网络铲（Web scraper，可类比考古用的洛阳铲）、网络蜘蛛（Web spider），其行为一般是先“爬”到对应的网页上，再把需要的信息“铲”下来。O’Reilly 这本书的封面图案是一只穿山甲，图灵公司把这本书的中文版定名为“Python 网络数据采集”。当我们看完这本书的时候，觉得网络数据采集程序也像是一只辛勤采蜜的小蜜蜂，它飞到花（目标网页）上，采集花粉（需要的信息），经过处理（数据清洗、存储）变成蜂蜜（可用的数据）。网络数据采集可以为生活加点儿蜜，亦如本书作者所说，“网络数据采集是为普通大众所喜闻乐见的计算机巫术”。

网络数据采集大有所为。在大数据深入人心的时代，网络数据采集作为网络、数据库与机器学习等领域的交汇点，已经成为满足个性化网络数据需求的最佳实践。搜索引擎可以满足人们对数据的共性需求，即“我来了，我看见”，而网络数据采集技术可以进一步精炼数据，把网络中杂乱无章的数据聚合成合理规范的形式，方便分析与挖掘，真正实现“我征服”。工作中，你可能经常为找数据而烦恼，或者眼睁睁看着眼前的几百页数据却只能长恨咫尺天涯，又或者数据杂乱无章的网站中满是带有陷阱的表单和坑爹的验证码，甚至需要的数据都在网页版的 PDF 和网络图片中。而作为一名网站管理员，你也需要了解常用的网络数据采集手段，以及常用的网络表单安全措施，以提高网站访问的安全性，所谓道高一尺，魔高一丈……一念清净，烈焰成池，一念觉醒，方登彼岸，本书试图成为解决这些问题的一念，让你茅塞顿开，船登彼岸。

网络数据采集并不是一门语言的独门秘籍，Python、Java、PHP、C#、Go 等语言都可以讲出精彩的故事。有人说编程语言就是宗教，不同语言的设计哲学不同，行为方式各异，“非我族类，其心必异”，但本着美好生活、快乐修行的初衷，我们对所有语言都时刻保持敬畏之心，尊重信仰自由，努力做好自己的功课。对爱好 Python 的人来说，人生苦短，Python 当歌！简洁轻松的语法，开箱即用的模块，强大快乐的社区，总可以快速构建出简单高效的解决方案。使用 Python 的日子总是充满快乐的，本书关于 Python 网络数

据采集的故事也不例外。网络数据采集涉及多个领域，内容包罗万象，因此本书覆盖的主题较多，涉及的知识面相对广阔，书中介绍的 Python 模块有 urllib、BeautifulSoup、lxml、Scrapy、PdfMiner、Requests、Selenium、NLTK、Pillow、unittest、PySocks 等，还有一些知名网站的 API、MySQL 数据库、OpenRefine 数据分析工具、PhanthomJS 无头浏览器以及 Tor 代理服务器等内容。每行到一处，皆是风景独好，而且作者也为每一个主题提供了深入研究的参考资料。不过，本书关于多进程（multiprocessing）、并发（concurrency）、集群（cluster）等高性能采集主题着墨不多，更加关注性能的读者，可以参考其他关于 Python 高性能和多核编程的书籍。总之，本书通俗易懂，简单易行，有编程基础的同学都可以阅读。不会 Python ？抽一节课时间学一下吧。

网络数据采集也应该有所不为。国内外关于网络数据保护的法律法规都在不断地制定与完善中，本书作者在书中介绍了美国与网络数据采集相关的法律与典型案例，呼吁网络爬虫严格控制网络数据采集的速度，降低被采集网站服务器的负担。恶意消耗别人网站的服务器资源，甚至拖垮别人网站是一件不道德的事情。众所周知，这已经不仅仅是一句“吸烟有害健康”之类的空洞口号，它可能导致更严重的法律后果，且行且珍惜！

语言是思想的解释器，书籍是语言的载体。本书英文原著是作者用英文解释器为自己思想写的载体，而译本是译者根据英文原著以及与作者的交流，用简体中文解释器为作者思想写的载体。读者拿到的中译本，是作者思想经过两层解释器转换的结果，其目的是希望帮助中文读者消除语言障碍，理解作者的思想，与作者产生共鸣，一起面对作者曾经遇到的问题，共同探索解决问题的方法，从而帮助读者提高解决问题的能力，增强直面 bug 的信心。bug 是产品生命中的挑战，好产品是不断面对 bug 并战胜 bug 的结果。译者水平有限，译文 bug 也在所难免，翻译有不到之处，还请各位读者批评指正！

最后要感谢图灵公司朱巍老师的大力支持，让译作得以顺利出版。也要感谢神烦小宝的温馨陪伴，每天 6 点叫我们起床，让业余时间格外宽裕。

译者联系方式——
邮箱：muxuezi@gmail.com，微信号：muxuezi
邮箱：carrieforchen@gmail.com，微信号：陈小莉

陶俊杰

2015 年 10 月

前言

对那些没有学过编程的人来说，计算机编程看着就像变魔术。如果编程是魔术（magic），那么网络数据采集（Web scraping）就是巫术（wizardry）；也就是运用“魔术”来实现精彩实用却又不费吹灰之力的“壮举”。

说句实话，在我的软件工程师职业生涯中，我几乎没有发现像网络数据采集这样的编程实践，可以同时吸引程序员和门外汉的注意。虽然写一个简单的网络爬虫并不难，就是先收集数据，再显示到命令行或者存储到数据库里，但是无论你之前已经做过多少次了，这件事永远会让你感到兴奋，同时又有新的可能。

不过遗憾的是，当和别的程序员提起网络数据采集时，我听到了很多关于这件事的误解与困惑。有些人不确定它是不是合法的（其实合法），有人不明白怎么处理那些到处都是 JavaScript、多媒体和 cookie 的新式网站，还有人对 API 和网络爬虫的区别感到困惑。

这本书的初衷是要解决人们对网络数据采集的诸多问题与误解，并对常见的网络数据采集任务提供全面的指导。

从第 1 章开始，我将不断地提供代码示例来演示书中内容。这些代码示例是开源的，无论注明出处与否都可以免费使用（但若注明会让作者感激不尽）。所有的代码示例都在 GitHub 网站上（https://github.com/REMitchell/python-scraping），可以查看和下载。

什么是网络数据采集

在互联网上进行自动数据采集这件事和互联网存在的时间差不多一样长。虽然网络数据采集并不是新术语，但是多年以来，这件事更常见的称谓是网页抓屏（screen scraping）、数据挖掘（data mining）、网络收割（Web harvesting）或其他类似的版本。今天大众好像更倾向于用“网络数据采集”，因此我在本书中使用这个术语，不过有时会把网络数据采集程序称为网络机器人（bots）。

理论上，网络数据采集是一种通过多种手段收集网络数据的方式，不光是通过与 API 交互（或者直接与浏览器交互）的方式。最常用的方法是写一个自动化程序向网络服务器请求数据（通常是用 HTML 表单或其他网页文件），然后对数据进行解析，提取需要的信息。

实践中，网络数据采集涉及非常广泛的编程技术和手段，比如数据分析、信息安全等。本书将在第一部分介绍关于网络数据采集和网络爬行（crawling）的基础知识，一些高级主题放在第二部分介绍。

为什么要做网络数据采集

如果你上网的唯一方式就是用浏览器，那么你其实失去了很多种可能。虽然浏览器可以更方便地执行 JavaScript，显示图片，并且可以把数据展示成更适合人类阅读的形式，但是网络爬虫收集和处理大量数据的能力更为卓越。不像狭窄的显示器窗口一次只能让你看一个网页，网络爬虫可以让你一次查看几千甚至几百万个网页。

另外，网络爬虫可以完成传统搜索引擎不能做的事情。用 Google 搜索“飞往波士顿最便宜的航班”，看到的是大量的广告和主流的航班搜索网站。Google 只知道这些网站的网页会显示什么内容，却不知道在航班搜索应用中输入的各种查询的准确结果。但是，设计较好的网络爬虫可以通过采集大量的网站数据，做出飞往波士顿航班价格随时间变化的图表，告诉你买机票的最佳时间。

你可能会问：“数据不是可以通过 API 获取吗？”（如果你不熟悉 API，请阅读第 4 章。）确实，如果你能找到一个可以解决你的问题的 API，那会非常给力。它们可以非常方便地向用户提供服务器里格式完好的数据。当你使用像 Twitter 或维基百科的 API 时，会发现一个 API 同时提供了不同的数据类型。通常，如果有 API 可用，API 确实会比写一个网络爬虫程序来获取数据更加方便。但是，很多时候你需要的 API 并不存在，这是因为：

- 你要收集的数据来自不同的网站，没有一个综合多个网站数据的 API；
- 你想要的数据非常小众，网站不会为你单独做一个 API；
- 一些网站没有基础设施或技术能力去建立 API。

即使 API 已经存在，可能还会有请求内容和次数限制，API 能够提供的数据类型或者数据格式可能也无法满足你的需求。

这时网络数据采集就派上用场了。你在浏览器上看到的内容，大部分都可以通过编写 Python 程序来获取。如果你可以通过程序获取数据，那么就可以把数据存储到数据库里。如果你可以把数据存储到数据库里，自然也就可以将这些数据可视化。

显然，大量的应用场景都会需要这种几乎可以毫无阻碍地获取数据的手段：市场预测、机器语言翻译，甚至医疗诊断领域，通过对新闻网站、文章以及健康论坛中的数据进行采集

和分析，也可以获得很多好处。

甚至在艺术领域，网络数据采集也为艺术创作开辟了新方向。由 Jonathan Harris 和 Sep Kamvar 在 2006 年发起的“我们感觉挺好”（We Feel Fine，http://wefeelfine.org/）项目，从大量英文博客中抓取许多以“I feel”和“I am feeling”开头的短句，最终做成了一个很受大众欢迎的数据可视图，描述了这个世界每天、每分钟的感觉。

无论你现在处于哪个领域，网络数据采集都可以让你的工作更高效，帮你提升生产力，甚至开创一个全新的领域。

关于本书

本书不仅介绍了网络数据采集，也为采集新式网络中的各种数据类型提供了全面的指导。虽然本书用的是 Python 编程语言，里面涉及 Python 的许多基础知识，但这并不是一本 Python 入门图书。

如果你不太懂编程，也完全不了解 Python，那么这本书看起来可能有点儿费劲。但是，如果你懂编程，那么书中的内容可以很快上手。附录 A 介绍了 Python 3.x 版本的安装和使用方法，全书将使用这个版本的 Python。如果你的电脑里只装了 Python 2.x 版本，可能需要先看看附录 A。

如果你想更全面地学习 Python，Bill Lubanovic 写的《Python 语言及其应用》[1] 是本非常好的教材，只是书有点儿厚。如果不想看书，Jessica McKellar 的教学视频 Introduction to Python（http://shop.oreilly.com/product/110000448.do）也非常不错。

附录 C 介绍并分析了几个商业案例以及犯罪事件，可以帮助你了解如何在美国合法地运行网络爬虫并使用数据。

技术书通常都是介绍一种语言或技术，而网络数据采集是一个比较综合的主题，涉及数据库、网络服务器、HTTP 协议、HTML 语言、网络安全、图像处理、数据科学等内容。本书尝试涵盖网络数据采集的所有内容。

第一部分深入讲解网络数据采集和网络爬行相关内容，并重点介绍全书都要用到的几个 Python 库。这部分内容可以看成这些库和技术的综合参考（对于一些特殊情形，后面会提供其他参考资料）。

第二部分介绍读者在动手编写网络爬虫的过程中可能会涉及的一些主题。不过，这些主题的范围特别广泛，这部分内容也不足以道尽玄机。因此，文中提供了许多常用的参考资料来补充更多的信息。

注 1：中文版已经由人民邮电出版社出版。——编者注

本书结构组织灵活，便于你直接跳到感兴趣的章节中阅读相应的网络数据采集技术。如果一个概念或一段代码在之前的章节中出现过，那么我会明确标注出具体的位置。

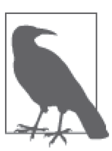
排版约定

本书使用了下列排版约定。

- 楷体
  表示新术语。
- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。
- 加粗等宽字体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。
- 斜体等宽字体（*`constant width italic`*）
  表示应该替换成用户输入的值，或根据上下文替换的值。



该图标表示提示或建议。



该图标表示一般性说明。



该图标表示警告或警示。

使用代码示例

补充材料（代码示例、练习等）可以从 https://github.com/REMitchell/python-scraping 下载。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无需联系我们获得许可。比如，用本书

的几个代码片段写一个程序就无需获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无需获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如："*Web Scraping with Python* by Ryan Mitchell (O'Reilly). Copyright 2015 Ryan Mitchell, 978-1-491-91029-0."

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://oreil.ly/1ePG2Uj

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

和那些基于海量用户反馈诞生的优秀产品一样，如果没有许多协作者、支持者和编辑的帮助，本书可能永远都不会出版。首先要感谢 O’Reilly 团队对这个小众主题图书的大力支持，感谢我的朋友和家人阅读初稿并提出宝贵的建议，还要感谢和我一起在 LinkeDrive 奋战的同事们帮我分担了很多工作。

尤其要感谢 Allyson MacDonald、Brian Anderson、Miguel Grinberg 和 Eric VanWyk 的建议、指导和偶尔的爱之深责之切。有一些章节和代码示例是根据他们的建议写成的。

还要感谢 Yale Specht 过去九个月用无尽的耐心和鼓励促成了这个项目，并在我的写作过程中对文体提出了宝贵的建议。没有他，这本书可能只用一半时间就能写完，但是不会像现在这么实用。

最后，要感谢 Jim Waldo，是他许多年前给一个小孩邮寄了一个 Linux 机箱和 *The Art and Science of C* 那本书，帮她开启了计算机世界的大门。

第一部分

创建爬虫

这部分内容重点介绍网络数据采集的基本原理：如何用 Python 从网络服务器请求信息，如何对服务器的响应进行基本处理，以及如何以自动化手段与网站进行交互。最终，你将轻松游弋于网络空间，创建出具有域名切换、信息收集以及信息存储功能的爬虫。

说实话，如果你想以较少的预先投入获取较高的回报，网络数据采集肯定是一个值得踏入的神奇领域。大体上，你遇到的 90% 的网络数据采集项目使用的都是接下来的六章里介绍的技术。这部分内容涵盖了一般人（也包括技术达人）在思考“网络爬虫”时通常的想法：

- 通过网站域名获取 HTML 数据
- 根据目标信息解析数据
- 存储目标信息
- 如果有必要，移动到另一个网页重复这个过程

这将为你学习本书第二部分中更复杂的项目奠定坚实的基础。不要天真地认为这部分内容没有第二部分里的一些比较高级的项目重要。其实，当你写自己的网络爬虫时，几乎每天都要用到第一部分的所有内容。

第 1 章

初见网络爬虫

一旦你开始采集网络数据，就会感受到浏览器为我们做的所有细节。网络上如果没有 HTML 文本格式层、CSS 样式层、JavaScript 执行层和图像渲染层，乍看起来会有点儿吓人，但是在这一章和下一章，我们将介绍如何不通过浏览器的帮助来格式化和理解数据。

本章将首先向网络服务器发送 GET 请求以获取具体网页，再从网页中读取 HTML 内容，最后做一些简单的信息提取，将我们要寻找的内容分离出来。

1.1 网络连接

如果你没在网络或网络安全上花过太多时间，那么互联网的原理可能看起来有点儿神秘。准确地说，每当打开浏览器连接 http://google.com 的时候，我们不会思考网络正在做什么，而且如今也不必思考。实际上，我认为很神奇的是，计算机接口已经如此先进，让大多数人上网的时候完全不思考网络是如何工作的。

但是，网络数据采集需要抛开一些接口的遮挡，不仅是在浏览器层（它如何解释所有的 HTML、CSS 和 JavaScript），有时也包括网络连接层。

我们通过下面的例子让你对浏览器获取信息的过程有一个基本的认识。Alice 有一台网络服务器。Bob 有一个台式机正准备连接 Alice 的服务器。当一台机器想与另一台机器对话时，下面的某个行为将会发生。

1. Bob 的电脑发送一串 1 和 0 比特值，表示电路上的高低电压。这些比特构成了一种信息，包括请求头和消息体。请求头包含当前 Bob 的本地路由器 MAC 地址和 Alice 的 IP

地址。消息体包含 Bob 对 Alice 服务器应用的请求。

2. Bob 的本地路由器收到所有 1 和 0 比特值，把它们理解成一个数据包（packet），从 Bob 自己的 MAC 地址"寄到"Alice 的 IP 地址。他的路由器把数据包"盖上"自己的 IP 地址作为"发件"地址，然后通过互联网发出去。
3. Bob 的数据包游历了一些中介服务器，沿着正确的物理 / 电路路径前进，到了 Alice 的服务器。
4. Alice 的服务器在她的 IP 地址收到了数据包。
5. Alice 的服务器读取数据包请求头里的目标端口（通常是网络应用的 80 端口，可以理解成数据包的"房间号"，IP 地址就是"街道地址"），然后把它传递到对应的应用——网络服务器应用上。
6. 网络服务器应用从服务器处理器收到一串数据，数据是这样的：
   - 这是一个 GET 请求
   - 请求文件 index.html
7. 网络服务器应用找到对应的 HTML 文件，把它打包成一个新的数据包发送给 Bob，然后通过它的本地路由器发出去，用同样的过程回传到 Bob 的机器上。

瞧！我们就这样实现了互联网。

那么，在这场数据交换中，网络浏览器从哪里开始参与的？完全没有参与。其实，在互联网的历史中，浏览器是一个比较年轻的发明，始于 1990 年的 Nexus 浏览器。

的确，网络浏览器是一个非常有用的应用，它创建信息的数据包，发送它们，然后把你获取的数据解释成漂亮的图像、声音、视频和文字。但是，网络浏览器就是代码，而代码是可以分解的，可以分解成许多基本组件，可重写、重用，以及做成我们想要的任何东西。网络浏览器可以让服务器发送一些数据，到那些对接无线（或有线）网络接口的应用上，但是许多语言也都有实现这些功能的库文件。

让我们看看 Python 是如何实现的：

```
from urllib.request import urlopen
html = urlopen("http://pythonscraping.com/pages/page1.html")
print(html.read())
```

你可以把这段代码保存为 scrapetest.py，然后在终端里运行如下命令：

```
$python scrapetest.py
```

注意，如果你的设备上安装了 Python 2.x，可能需要直接指明版本才能运行 Python 3.x 代码：

```
$python3 scrapetest.py
```

这将会输出 http://pythonscraping.com/pages/page1.html 这个网页的全部 HTML 代码。更准确地说，这会输出在域名为 http://pythonscraping.com 的服务器上 < 网络应用根地址 >/pages 文件夹里的 HTML 文件 page1.html 的源代码。

有什么区别？现在大多数网页需要加载许多相关的资源文件。可能是图像文件、JavaScript 文件、CSS 文件，或你需要连接的其他各种网页内容。当网络浏览器遇到一个标签时，比如 `<img src="cuteKitten.jpg">`，会向服务器发起另一个请求，以获取 cuteKitten.jpg 文件中的数据为用户充分渲染网页。但是，我们的 Python 程序没有返回并向服务器请求多个文件的逻辑，它只能读取我们已经请求的单个 HTML 文件。

那么我们应该怎样做呢？幸好 Python 语法接近正常英文，下面这行代码

```
from urllib.request import urlopen
```

其实已经显示了它的含义：它查找 Python 的 request 模块（在 urllib 库里面），只导入一个 `urlopen` 函数。

> **urllib 还是 urllib2？**
>
> 如果你用过 Python 2.x 里的 urllib2 库，可能会发现 urllib2 与 urllib 有些不同。在 Python 3.x 里，urllib2 改名为 urllib，被分成一些子模块：`urllib.request`、`urllib.parse` 和 `urllib.error`。尽管函数名称大多和原来一样，但是在用新的 urllib 库时需要注意哪些函数被移动到子模块里了。

`urllib` 是 Python 的标准库（就是说你不用额外安装就可以运行这个例子），包含了从网络请求数据，处理 cookie，甚至改变像请求头和用户代理这些元数据的函数。我们将在本书中广泛使用 `urllib`，所以建议你读读这个库的 Python 文档（https://docs.python.org/3/library/urllib.html）。

`urlopen` 用来打开并读取一个从网络获取的远程对象。因为它是一个非常通用的库（它可以轻松读取 HTML 文件、图像文件，或其他任何文件流），所以我们将在本书中频繁地使用它。

1.2 BeautifulSoup简介

> "美味的汤，绿色的浓汤，
> 在热气腾腾的盖碗里装！
> 谁不愿意尝一尝，这样的好汤？
> 晚餐用的汤，美味的汤！"

BeautifulSoup 库的名字取自刘易斯 · 卡罗尔在《爱丽丝梦游仙境》里的同名诗歌。在故事

中，这首诗是素甲鱼[1]唱的。

就像它在仙境中的说法一样，BeautifulSoup 尝试化平淡为神奇。它通过定位 HTML 标签来格式化和组织复杂的网络信息，用简单易用的 Python 对象为我们展现 XML 结构信息。

1.2.1 安装BeautifulSoup

由于 BeautifulSoup 库不是 Python 标准库，因此需要单独安装。在本书中，我们将使用最新的 BeautifulSoup 4 版本（也叫 BS4）。BeautifulSoup 4 的所有安装方法都在 http://www.crummy.com/software/BeautifulSoup/bs4/doc/ 里面。Linux 系统上的基本安装方法是：

```
$sudo apt-get install python-bs4
```

对于 Mac 系统，首先用

```
$sudo easy_install pip
```

安装 Python 的包管理器 pip，然后运行

```
$pip install beautifulsoup4
```

来安装库文件。

另外，注意如果你的设备同时安装了 Python 2.x 和 Python 3.x，你需要用 `python3` 运行 Python 3.x：

```
$python3 myScript.py
```

当你安装包的时候，如果有可能安装到了 Python 2.x 而不是 Python 3.x 里，就需要使用：

```
$sudo python3 setup.py install
```

如果用 pip 安装，你还可以用 `pip3` 安装 Python 3.x 版本的包：

```
$pip3 install beautifulsoup4
```

在 Windows 系统上安装与在 Mac 和 Linux 上安装差不多。从上面的下载链接下载最新的 BeautifulSoup 4 源代码，解压后进入文件，然后执行：

```
>python setup.py install
```

这样就可以了！ BeautifulSoup 将被当作设备上的一个 Python 库。你可以在 Python 终端里导入它测试一下：

注 1：Mock Turtle，它本身是一个双关语，指英国维多利亚时代的流行菜肴素甲鱼汤，其实不是甲鱼而是牛肉，如同中国的豆制品素鸡，名为素鸡，其实与鸡无关。

```
$python
> from bs4 import BeautifulSoup
```

如果没有错误，说明导入成功了。

另外，还有一个 Windows 版 pip（https://pypi.python.org/pypi/setuptools）的 .exe 格式安装器，装了之后你就可以轻松安装和管理包了：

```
>pip install beautifulsoup4
```

用虚拟环境保存库文件

如果你同时负责多个 Python 项目，或者想要轻松打包某个项目及其关联的库文件，再或者你担心已安装的库之间可能有冲突，那么你可以安装一个 Python 虚拟环境来分而治之。

当一个 Python 库不用虚拟环境安装的时候，你实际上是全局安装它。这通常需要有管理员权限，或者以 root 身份安装，这个库文件对设备上的每个用户和每个项目都是存在的。好在创建虚拟环境非常简单：

```
$ virtualenv scrapingEnv
```

这样就创建了一个叫作 scrapingEnv 的新环境，你需要先激活它再使用：

```
$ cd scrapingEnv/
$ source bin/activate
```

激活环境之后，你会发现环境名称出现在命令行提示符前面，提醒你当前处于虚拟环境中。后面你安装的任何库和执行的任何程序都是在这个环境下运行。

在新建的 scrapingEnv 环境里，可以安装并使用 BeautifulSoup：

```
(scrapingEnv)ryan$ pip install beautifulsoup4
(scrapingEnv)ryan$ python
> from bs4 import BeautifulSoup
>
```

当不再使用虚拟环境中的库时，可以通过释放命令来退出环境：

```
(scrapingEnv)ryan$ deactivate
ryan$ python
> from bs4 import BeautifulSoup
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ImportError: No module named 'bs4'
```

将项目关联的所有库单独放在一个虚拟环境里，还可以轻松打包整个环境发生给其他人。只要他们的 Python 版本和你的相同，你打包的代码就可以直接通过虚拟环境运行，不需要再安装任何库。

尽管本书的例子都不要求你使用虚拟环境，但是请记住，你可以在任何时候激活并使用它。

1.2.2 运行BeautifulSoup

BeautifulSoup 库最常用的对象恰好就是 BeautifulSoup 对象。让我们把本章开头的例子调整一下运行看看：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
html = urlopen("http://www.pythonscraping.com/pages/page1.html")
bsObj = BeautifulSoup(html.read())
print(bsObj.h1)
```

输出结果是：

```
<h1>An Interesting Title</h1>
```

和前面例子一样，我们导入 `urlopen`，然后调用 `html.read()` 获取网页的 HTML 内容。这样就可以把 HTML 内容传到 BeautifulSoup 对象，转换成下面的结构：

- **html** → <html><head>...</head><body>...</body></html>
  — **head** → <head><title>A Useful Page<title></head>
    — **title** → <title>A Useful Page</title>
  — **body** → <body><h1>An Int...</h1><div>Lorem ip...</div></body>
    — **h1** → <h1>An Interesting Title</h1>
    — **div** → <div>Lorem Ipsum dolor...</div>

可以看出，我们从网页中提取的 `<h1>` 标签被嵌在 BeautifulSoup 对象 `bsObj` 结构的第二层（html → body → h1）。但是，当我们从对象里提取 h1 标签的时候，可以直接调用它：

```
bsObj.h1
```

其实，下面的所有函数调用都可以产生同样的结果：

```
bsObj.html.body.h1
bsObj.body.h1
bsObj.html.h1
```

希望这个例子可以向你展示 `BeautifulSoup` 库的强大与简单。其实，任何 HTML（或 XML）文件的任意节点信息都可以被提取出来，只要目标信息的旁边或附近有标记就行。在第 3 章，我们将进一步探讨一些更复杂的 BeautifulSoup 函数，还会介绍正则表达式，以及如何把正则表达式用于 `BeautifulSoup` 以对网站信息进行提取。

1.2.3 可靠的网络连接

网络是十分复杂的。网页数据格式不友好，网站服务器宕机，目标数据的标签找不到，都是很麻烦的事情。网络数据采集最痛苦的遭遇之一，就是爬虫运行的时候你洗洗睡了，梦想着明天一早数据就都会采集好放在数据库里，结果第二天醒来，你看到的却是一个因某种数据格式异常导致运行错误的爬虫，在前一天当你不再盯着屏幕去睡觉之后，没过一会儿爬虫就不再运行了。那个时候，你可能想骂发明互联网（以及那些奇葩的网络数据格式）的人，但是你真正应该斥责的人是你自己，为什么一开始不估计可能会出现的异常！

让我们看看爬虫 `import` 语句后面的第一行代码，如何处理那里可能出现的异常：

```
html = urlopen("http://www.pythonscraping.com/pages/page1.html")
```

这行代码主要可能会发生两种异常：

- 网页在服务器上不存在（或者获取页面的时候出现错误）
- 服务器不存在

第一种异常发生时，程序会返回 HTTP 错误。HTTP 错误可能是“404 Page Not Found”“500 Internal Server Error”等。所有类似情形，`urlopen` 函数都会抛出“HTTPError”异常。我们可以用下面的方式处理这种异常：

```
try:
    html = urlopen("http://www.pythonscraping.com/pages/page1.html")
except HTTPError as e:
    print(e)
    # 返回空值,中断程序,或者执行另一个方案
else:
    # 程序继续。注意:如果你已经在上面异常捕捉那一段代码里返回或中断(break),
    # 那么就不需要使用else语句了,这段代码也不会执行
```

如果程序返回 HTTP 错误代码，程序就会显示错误内容，不再执行 `else` 语句后面的代码。

如果服务器不存在（就是说链接 http://www.pythonscraping.com/ 打不开，或者是 URL 链接写错了），`urlopen` 会返回一个 `None` 对象。这个对象与其他编程语言中的 `null` 类似。我们可以增加一个判断语句检测返回的 `html` 是不是 `None`：

```
if html is None:
    print("URL is not found")
else:
    # 程序继续
```

当然，即使网页已经从服务器成功获取，如果网页上的内容并非完全是我们期望的那样，仍然可能会出现异常。每当你调用 BeautifulSoup 对象里的一个标签时，增加一个检查条件保证标签确实存在是很聪明的做法。如果你想要调用的标签不存在，BeautifulSoup 就会返

回 None 对象。不过，如果再调用这个 None 对象下面的子标签，就会发生 AttributeError 错误。

下面这行代码（nonExistentTag 是虚拟的标签，BeautifulSoup 对象里实际没有）

```
print(bsObj.nonExistentTag)
```

会返回一个 None 对象。处理和检查这个对象是十分必要的。如果你不检查，直接调用这个 None 对象的子标签，麻烦就来了。如下所示。

```
print(bsObj.nonExistentTag.someTag)
```

这时就会返回一个异常：

```
AttributeError: 'NoneType' object has no attribute 'someTag'
```

那么我们怎么才能避免这两种情形的异常呢？最简单的方式就是对两种情形进行检查：

```
try:
    badContent = bsObj.nonExistingTag.anotherTag
except AttributeError as e:
    print("Tag was not found")
else:
    if badContent == None:
        print ("Tag was not found")
    else:
        print(badContent)
```

初看这些检查与错误处理的代码会觉得有点儿累赘，但是，我们可以重新简单组织一下代码，让它变得不那么难写（更重要的是，不那么难读）。例如，下面的代码是上面爬虫的另一种写法：

```
from urllib.request import urlopen
from urllib.error import HTTPError
from bs4 import BeautifulSoup
    def getTitle(url):
        try:
            html = urlopen(url)
        except HTTPError as e:
            return None
        try:
            bsObj = BeautifulSoup(html.read())
            title = bsObj.body.h1
        except AttributeError as e:
            return None
        return title
title = getTitle("http://www.pythonscraping.com/pages/page1.html")
if title == None:
    print("Title could not be found")
else:
    print(title)
```

在这个例子中，我们创建了一个 getTitle 函数，可以返回网页的标题，如果获取网页的时候遇到问题就返回一个 None 对象。在 getTitle 函数里面，我们像前面那样检查了 HTTPError，然后把两行 BeautifulSoup 代码封装在一个 try 语句里面。这两行中的任何一行有问题，AttributeError 都可能被抛出（如果服务器不存在，html 就是一个 None 对象，html.read() 就会抛出 AttributeError）。其实，我们可以在 try 语句里面放任意多行代码，或者放一个在任意位置都可以抛出 AttributeError 的函数。

在写爬虫的时候，思考代码的总体格局，让代码既可以捕捉异常又容易阅读，这是很重要的。如果你还希望能够很大程度地重用代码，那么拥有像 getSiteHTML 和 getTitle 这样的通用函数（具有周密的异常处理功能）会让快速稳定地网络数据采集变得简单易行。

第 2 章

复杂HTML解析

当米开朗基罗被问及如何完成《大卫》这样匠心独具的雕刻作品时，他有一段著名的回答："很简单，你只要用锤子把石头上不像大卫的地方敲掉就行了。"

虽然网络数据采集和大理石雕刻大相径庭，但是当我们从复杂的网页中寻觅信息时，也必须持有类似的态度。在我们找到目标信息之前，有很多技巧可以帮我们"敲掉"网页上那些不需要的信息。这一章我们将介绍解析复杂的 HTML 页面的方法，从中抽取出我们需要的信息。

2.1　不是一直都要用锤子

面对页面解析难题（Gordian Knot）的时候，不假思索地直接写几行语句来抽取信息是非常直接的做法。但是，像这样鲁莽放纵地使用技术，只会让程序变得难以调试或脆弱不堪，甚至二者兼具。在开始解析网页之前，让我们看一些在解析复杂的 HTML 页面时需要避免的问题。

假如你已经确定了目标内容，可能是采集一个名字、一组统计数据，或者一段文字。你的目标内容可能隐藏在一个 HTML"烂泥堆"的第 20 层标签里，带有许多没用的标签或 HTML 属性。假如你不经考虑地直接写出下面这样一行代码来抽取内容：

```
bsObj.findAll("table")[4].findAll("tr")[2].find("td").findAll("div")[1].find("a")
```

虽然也可以达到目标，但这样看起来并不是很好。除了代码欠缺美感之外，还有一个问题是，当网站管理员对网站稍作修改之后，这行代码就会失效，甚至可能会毁掉整个网络爬

虫。那么你应该怎么做呢？

- 寻找“打印此页”的链接，或者看看网站有没有 HTML 样式更友好的移动版（把自己的请求头设置成处于移动设备的状态，然后接收网站移动版，更多内容在第 12 章介绍）。
- 寻找隐藏在 JavaScript 文件里的信息。要实现这一点，你可能需要查看网页加载的 JavaScript 文件。我曾经要把一个网站上的街道地址（以经度和纬度呈现的）整理成格式整洁的数组时，查看过内嵌谷歌地图的 JavaScript 文件，里面有每个地址的标记点。
- 虽然网页标题经常会用到，但是这个信息也许可以从网页的 URL 链接里获取。
- 如果你要找的信息只存在于一个网站上，别处没有，那你确实是运气不佳。如果不只限于这个网站，那么你可以找找其他数据源。有没有其他网站也显示了同样的数据？网站上显示的数据是不是从其他网站上抓取后攒出来的？

尤其是在面对埋藏很深或格式不友好的数据时，千万不要不经思考就写代码，一定要三思而后行。如果你确定自己不能另辟蹊径，那么本章后面的内容就是为你准备的。

2.2 再端一碗BeautifulSoup

在第 1 章里，我们快速演示了 BeautifulSoup 的安装与运行过程，同时也实现了每次选择一个对象的解析方法。在这一节，我们将介绍通过属性查找标签的方法，标签组的使用，以及标签解析树的导航过程。

基本上，你见过的每个网站都会有层叠样式表（Cascading Style Sheet，CSS）。虽然你可能会认为，专门为了让浏览器和人类可以理解网站内容而设计一个展现样式的层，是一件愚蠢的事，但是 CSS 的发明却是网络爬虫的福音。CSS 可以让 HTML 元素呈现出差异化，使那些具有完全相同修饰的元素呈现出不同的样式。比如，有一些标签看起来是这样：

```
<span class="green"></span>
```

而另一些标签看起来是这样：

```
<span class="red"></span>
```

网络爬虫可以通过 `class` 属性的值，轻松地区分出两种不同的标签。例如，它们可以用 BeautifulSoup 抓取网页上所有的红色文字，而绿色文字一个都不抓。因为 CSS 通过属性准确地呈现网站的样式，所以你大可放心，大多数新式网站上的 `class` 和 `id` 属性资源都非常丰富。

下面让我们创建一个网络爬虫来抓取 http://www.pythonscraping.com/pages/warandpeace.html 这个网页。

在这个页面里，小说人物的对话内容都是红色的，人物名称都是绿色的。你可以看到网页

源代码里的 span 标签，引用了对应的 CSS 属性，如下所示：

```
"<span class="red">Heavens! what a virulent attack!</span>" replied <span class=
"green">the prince</span>, not in the least disconcerted by this reception.
```

我们可以抓出整个页面，然后创建一个 BeautifulSoup 对象，和第 1 章里使用的程序类似：

```python
from urllib.request import urlopen
from bs4 import BeautifulSoup
html = urlopen("http://www.pythonscraping.com/pages/warandpeace.html")
bsObj = BeautifulSoup(html)
```

通过 BeautifulSoup 对象，我们可以用 findAll 函数抽取只包含在 标签里的文字，这样就会得到一个人物名称的 Python 列表（findAll 是一个非常灵活的函数，我们后面会经常用到它）：

```python
nameList = bsObj.findAll("span", {"class":"green"})
for name in nameList:
    print(name.get_text())
```

代码执行以后就会按照《战争与和平》中的人物出场顺序显示所有的人名。这是怎么实现的呢？之前，我们调用 bsObj.tagName 只能获取页面中的第一个指定的标签。现在，我们调用 bsObj.findAll(tagName, tagAttributes) 可以获取页面中所有指定的标签，不再只是第一个了。

获取人名列表之后，程序遍历列表中所有的名字，然后打印 name.get_text()，就可以把标签中的内容分开显示了。

什么时候使用 get_text() 与什么时候应该保留标签？

.get_text() 会把你正在处理的 HTML 文档中所有的标签都清除，然后返回一个只包含文字的字符串。假如你正在处理一个包含许多超链接、段落和标签的大段源代码，那么 .get_text() 会把这些超链接、段落和标签都清除掉，只剩下一串不带标签的文字。

用 BeautifulSoup 对象查找你想要的信息，比直接在 HTML 文本里查找信息要简单得多。通常在你准备打印、存储和操作数据时，应该最后才使用 .get_text()。一般情况下，你应该尽可能地保留 HTML 文档的标签结构。

2.2.1 BeautifulSoup的find()和findAll()

BeautifulSoup 里的 find() 和 findAll() 可能是你最常用的两个函数。借助它们，你可以通过标签的不同属性轻松地过滤 HTML 页面，查找需要的标签组或单个标签。

这两个函数非常相似，BeautifulSoup 文档里两者的定义就是这样：

```
findAll(tag, attributes, recursive, text, limit, keywords)
find(tag, attributes, recursive, text, keywords)
```

很可能你会发现，自己在 95% 的时间里都只需要使用前两个参数：tag 和 attributes。但是，我们还是应该仔细地观察所有的参数。

标签参数 tag 前面已经介绍过——你可以传一个标签的名称或多个标签名称组成的 Python 列表做标签参数。例如，下面的代码将返回一个包含 HTML 文档中所有标题标签的列表：[1]

```
.findAll({"h1","h2","h3","h4","h5","h6"})
```

属性参数 attributes 是用一个 Python 字典封装一个标签的若干属性和对应的属性值。例如，下面这个函数会返回 HTML 文档里红色与绿色两种颜色的 span 标签：

```
.findAll("span", {"class":{"green", "red"}})
```

递归参数 recursive 是一个布尔变量。你想抓取 HTML 文档标签结构里多少层的信息？如果 recursive 设置为 True，findAll 就会根据你的要求去查找标签参数的所有子标签，以及子标签的子标签。如果 recursive 设置为 False，findAll 就只查找文档的一级标签。findAll 默认是支持递归查找的（recursive 默认值是 True）；一般情况下这个参数不需要设置，除非你真正了解自己需要哪些信息，而且抓取速度非常重要，那时你可以设置递归参数。

文本参数 text 有点不同，它是用标签的文本内容去匹配，而不是用标签的属性。假如我们想查找前面网页中包含“the prince”内容的标签数量，我们可以把之前的 findAll 方法换成下面的代码：

```
nameList = bsObj.findAll(text="the prince")
print(len(nameList))
```

输出结果为“7”。

范围限制参数 limit，显然只用于 findAll 方法。find 其实等价于 findAll 的 limit 等于 1 时的情形。如果你只对网页中获取的前 *x* 项结果感兴趣，就可以设置它。但是要注意，这个参数设置之后，获得的前几项结果是按照网页上的顺序排序的，未必是你想要的那前几项。

还有一个关键词参数 keyword，可以让你选择那些具有指定属性的标签。例如：

```
allText = bsObj.findAll(id="text")
print(allText[0].get_text())
```

注 1：如果你想获得文档里的一组 h<some_level> 标签，可以用更简洁的方法写代码来完成。我们将在 2.4 节介绍这类问题的处理方法。

关键词参数的注意事项

虽然关键词参数 keyword 在一些场景中很有用，但是，它是 BeautifulSoup 在技术上做的一个冗余功能。任何用关键词参数能够完成的任务，同样可以用本章后面将介绍的技术解决（请参见 2.3 节和 2.6 节）。

例如，下面两行代码是完全一样的：

```
bsObj.findAll(id="text")
bsObj.findAll("", {"id":"text"})
```

另外，用 keyword 偶尔会出现问题，尤其是在用 class 属性查找标签的时候，因为 class 是 Python 中受保护的关键字。也就是说，class 是 Python 语言的保留字，在 Python 程序里是不能当作变量或参数名使用的（和前面介绍的 BeautifulSoup.findAll() 里的 keyword 无关）[2]。假如你运行下面的代码，Python 就会因为你误用 class 保留字而产生一个语法错误：

```
bsObj.findAll(class="green")
```

不过，你可以用 BeautifulSoup 提供的有点儿臃肿的方案，在 class 后面增加一个下划线：

```
bsObj.findAll(class_="green")
```

另外，你也可以用属性参数把 class 用引号包起来：

```
bsObj.findAll("", {"class":"green"})
```

看到这里，你可能会扪心自问："现在我是不是已经知道如何用标签属性获取一组标签了——用字典把属性传到函数里就行了？"

回忆一下前面的内容，通过标签参数 tag 把标签列表传到 .findAll() 里获取一列标签，其实就是一个"或"关系的过滤器（即选择所有带标签 1 或标签 2 或标签 3……的一列标签）。如果你的标签列表很长，就需要花很长时间才能写完。而关键词参数 keyword 可以让你增加一个"与"关系的过滤器来简化工作。

2.2.2　其他BeautifulSoup对象

看到这里，你已经见过 BeautifulSoup 库里的两种对象了。

- BeautifulSoup 对象
  前面代码示例中的 bsObj
- 标签 Tag 对象
  BeautifulSoup 对象通过 find 和 findAll，或者直接调用子标签获取的一列对象或单个对象，就像：

注 2：Python 语言参考里提供了关键词列表（https://docs.python.org/3/reference/lexical_analysis.html#keywords）。

```
bsObj.div.h1
```

但是，这个库还有另外两种对象，虽然不常用，却应该了解一下。

- `NavigableString` 对象
  用来表示标签里的文字，不是标签（有些函数可以操作和生成 `NavigableString` 对象，而不是标签对象）。

- `Comment` 对象
  用来查找 HTML 文档的注释标签，`<!-- 像这样 -->`

这四个对象是你用 BeautifulSoup 库时会遇到的所有对象（写作本书的时候）。

2.2.3　导航树

`findAll` 函数通过标签的名称和属性来查找标签 。但是如果你需要通过标签在文档中的位置来查找标签，该怎么办？这就是导航树（Navigating Trees）的作用。在第 1 章里，我们看过用单一方向进行 BeautifulSoup 标签树的导航：

```
bsObj.tag.subTag.anotherSubTag
```

现在我们用虚拟的在线购物网站 http://www.pythonscraping.com/pages/page3.html 作为要抓取的示例网页，演示 HTML 导航树的纵向和横向导航（如图 2-1 所示）。

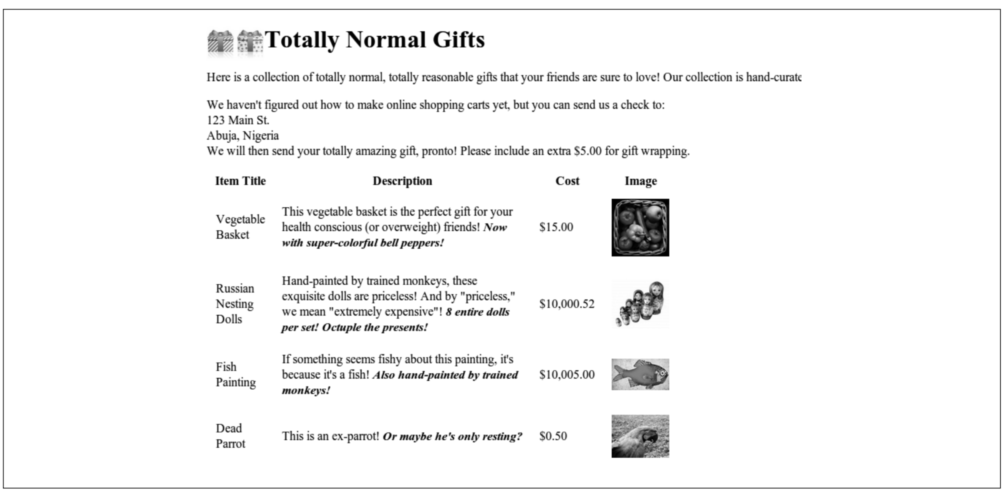


图 2-1：http://www.pythonscraping.com/pages/page3.html 截图

这个 HTML 页面可以映射成一棵树（为了简洁，省略了一些标签），如下所示：

- html
  — body
    — div.wrapper
      — h1
      — div.content
      — table#giftList
        — tr
          — th
          — th
          — th
          — th
        — tr.gift#gift1
          — td
          — td
            — span.excitingNote
          — td
          — td
            — img
        — ……其他表格行省略了……
    — div.footer

在后面几节内容里，我们仍然以这个 HTML 标签结构为例。

1. 处理子标签和其他后代标签

在计算机科学和一些数学领域中，你经常会听到“虐子”事件（比喻对一些子事件的处理方式）：移动它们，储存它们，删除它们，甚至杀死它们。值得庆幸的是，在 BeautifulSoup 里，子标签的处理方式没那么残忍。

和许多其他库一样，在 BeautifulSoup 库里，*孩子*（child）和*后代*（descendant）有显著的不同：和人类的家谱一样，子标签就是一个父标签的下一级，而后代标签是指一个父标签下面所有级别的标签。例如，`tr` 标签是 `tabel` 标签的子标签，而 `tr`、`th`、`td`、`img` 和 `span` 标签都是 `tabel` 标签的后代标签（我们的示例页面中就是如此）。所有的子标签都是后代标签，但不是所有的后代标签都是子标签。

一般情况下，BeautifulSoup 函数总是处理当前标签的后代标签。例如，`bsObj.body.h1` 选择了 `body` 标签后代里的第一个 `h1` 标签，不会去找 `body` 外面的标签。

类似地，`bsObj.div.findAll("img")` 会找出文档中第一个 `div` 标签，然后获取这个 `div` 后代里所有的 `img` 标签列表。

如果你只想找出子标签，可以用 .children 标签：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen("http://www.pythonscraping.com/pages/page3.html")
bsObj = BeautifulSoup(html)

for child in bsObj.find("table",{"id":"giftList"}).children:
    print(child)
```

这段代码会打印 giftList 表格中所有产品的数据行。如果你用 descendants() 函数而不是 children() 函数，那么就会有二十几个标签打印出来，包括 img 标签、span 标签，以及每个 td 标签。掌握子标签与后代标签的差别十分重要！

2. 处理兄弟标签

BeautifulSoup 的 next_siblings() 函数可以让收集表格数据成为简单的事情，尤其是处理带标题行的表格：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
html = urlopen("http://www.pythonscraping.com/pages/page3.html")
bsObj = BeautifulSoup(html)

for sibling in bsObj.find("table",{"id":"giftList"}).tr.next_siblings:
    print(sibling)
```

这段代码会打印产品列表里的所有行的产品，第一行表格标题除外。为什么标题行被跳过了呢？有两个理由。首先，对象不能把自己作为兄弟标签。任何时候你获取一个标签的兄弟标签，都不会包含这个标签本身。其次，这个函数只调用后面的兄弟标签。例如，如果我们选择一组标签中位于中间位置的一个标签，然后用 next_siblings() 函数，那么它就只会返回在它后面的兄弟标签。因此，选择标签行然后调用 next_siblings，可以选择表格中除了标题行以外的所有行。

让标签的选择更具体

如果我们选择 bsObj.table.tr 或直接就用 bsObj.tr 来获取表格中的第一行，上面的代码也可以获得正确的结果。但是，我们还是采用更长的形式写了一行代码，这可以避免各种意外：

```
bsObj.find("table",{"id":"giftList"}).tr
```

即使页面上只有一个表格（或其他目标标签），只用标签也很容易丢失细节。另外，页面布局总是不断变化的。一个标签这次是在表格中第一行的位置，没准儿哪天就在第二行或第三行了。如果想让你的爬虫更稳定，最好还是让标签的选择更加具体。如果有属性，就利用标签的属性。

和 next_siblings 一样，如果你很容易找到一组兄弟标签中的最后一个标签，那么 previous_siblings 函数也会很有用。

当然，还有 next_sibling 和 previous_sibling 函数，与 next_siblings 和 previous_siblings 的作用类似，只是它们返回的是单个标签，而不是一组标签。

3. 父标签处理

在抓取网页的时候，查找父标签的需求比查找子标签和兄弟标签要少很多。通常情况下，如果以抓取网页内容为目的来观察 HTML 页面，我们都是从最上层标签开始的，然后思考如何定位我们想要的数据块所在的位置。但是，偶尔在特殊情况下你也会用到 BeautifulSoup 的父标签查找函数，parent 和 parents。例如：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen("http://www.pythonscraping.com/pages/page3.html")
bsObj = BeautifulSoup(html)
print(bsObj.find("img",{"src":"../img/gifts/img1.jpg"
                       }).parent.previous_sibling.get_text())
```

这段代码会打印 ../img/gifts/img1.jpg 这个图片对应商品的价格（这个示例中价格是 $15.00）。

这是如何实现的呢？下面的图形是我们正在处理的 HTML 页面的部分结构，用数字表示步骤的话：

- <tr>
  — <td>
  — <td>
  — <td>(3)
    — "$15.00" (4)
  — <td>(2)
    — <img src="../img/gifts/img1.jpg"> (1)

(1) 选择图片标签 src="../img/gifts/img1.jpg"；
(2) 选择图片标签的父标签（在示例中是 <td> 标签）；
(3) 选择 <td> 标签的前一个兄弟标签 previous_sibling（在示例中是包含美元价格的 <td> 标签）；
(4) 选择标签中的文字，“$15.00”。

2.3 正则表达式

计算机科学里曾经有个笑话：“如果你有一个问题打算用正则表达式（regular expression）

来解决，那么就是两个问题了。”

不幸的是，正则表达式（通常简写 regex）经常被嘲笑是一堆随机符号的混和物，看着毫无意义。这种印象让人对其避而远之，然后费尽心思写一堆没必要又复杂的查找和过滤函数，其实他们真正需要的就是一行正则表达式。

其实正则表达式上手一点儿也不难，而且运行很快，通过一些简单的例子就可以轻松地学会。

之所以叫正则表达式，是因为它们可以识别正则字符串（regular string）；也就是说，它们可以这么定义：“如果你给我的字符串符合规则，我就返回它”，或者是“如果字符串不符合规则，我就忽略它”。这在要求快速浏览大文档，以查找像电话号码和邮箱地址之类的字符串时是非常方便的。

注意这里我用了一个词组正则字符串。什么是正则字符串？其实就是任意可以用一系列线性规则构成的字符串[3]，就像：

(1) 字母“a”至少出现一次；
(2) 后面跟着字母“b”重复 5 次；
(3) 后面再跟字母“c”重复任意偶数次；
(4) 最后一位是字母“d”，也可以没有。

满足上面规则的字符串有：“aaaabbbbbccccd”“aabbbbbcc”等（有无穷多种变化）。

正则表达式就是表达这组规则的缩写。这组规则的正则表达式如下所示：

```
aa*bbbbb(cc)*(d | )
```

第一次看这个字符串会觉得有点儿奇葩，但是当我们把它分解之后就会很清楚了。

- *aa**
  a 后面跟着的 a*（读作 a 星）表示“重复任意次 a，包括 0 次”。这样就可以保证字母 a 至少出现一次。

- *bbbbb*
  这没有什么特别的——就是 5 次 b。

- *(cc)**
  任意偶数个字符都可以编组，这个规则是用括号两个 c，然后后面跟一个星号，表示有任意次两个 c（也可以是 0 次）。

注 3：你可能会问：“有没有‘非正则’的表达式？”非正则表达式超出了本书的介绍范围，它们其实是指那些像“前面是素数个 a，后面跟着两倍于 a 数量的 b”“写一个回文”之类的字符串。用正则表达式不可能写出这类字符串。不过好在我的网络爬虫至今还从来没有遇到过这种需求。

- *(d|)*

  增加一个竖线（|）在表达式里表示“这个或那个”。本例是表示“增加一个后面跟着空格的 d，或者只有一个空格”。这样我们可以保证字符串的结尾最多是一个后面跟着空格的 d。

> **尝试正则表达式**
>
> 在学习书写正则表达式的时候，做一些实验感受一下它们如何工作，这是至关重要的。
>
> 如果你不想打开代码编辑器，写完再运行程序检查正则表达式的运行是否符合预期，那么你可以去 RegexPal（http://regexpal.com/）这类网站上在线测试正则表达式。

正则表达式在实际中的一个经典应用是识别邮箱地址。虽然不同邮箱服务器的邮箱地址的具体规则不尽相同，但是我们还是可以创建几条通用规则。每条规则对应的正则表达式如下表第 2 列所示。

规　　则	正则表达式
1. 邮箱地址的第一部分至少包括一种内容：大写字母、小写字母、数字 0~9、点号（.）、加号（+）或下划线（_）	[A-Za-z0-9\._+]+：这个正则表达式简写非常智慧。例如，它用“A-Z”表示“任意 A~Z 的大写字母”。把所有可能的序列和符号放在中括号（不是小括号）里表示“括号中的符号里任何一个”。要注意后面的加号，它表示“这些符号都可以出现多次，且至少出现 1 次”
2. 之后，邮箱地址会包含一个 @ 符号	@：这个符号很直接。@ 符号必须出现在中间位置，有且仅有 1 次
3. 在符合 @ 之后，邮箱地址还必须至少包含一个大写或小写字母	[A-Za-z]+：可能只在域名的前半部分、符号 @ 后面用字母。而且，至少有一个字母
4. 之后跟一个点号（.）	\.：在域名前必须有一个点号（.）
5. 最后邮箱地址用 com、org、edu、net 结尾（实际上，顶级域名有很多种可能，但是作为示例演示这四个后缀够用了）。	(com|org|edu|net)：这样列出了邮箱地址中可能出现在点号之后的字母序列

把上面的规则连接起来，就获得了完整的正则表达式：

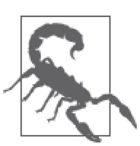
```
[A-Za-z0-9\._+]+@[A-Za-z]+\.(com|org|edu|net)
```

当我们动手开始写正则表达式的时候，最好先写一个步骤列表描述出你的目标字符串结构。还要注意一些细节的处理。比如，当你识别电话号码的时候，会考虑国家代码和分机号吗？

表 2-1 用简单的说明和例子列举了正则表达式的一些常用符号。这个列表并不是全部符号，另外就像之前所说的，可能在不同编程语言中会遇到一些变化。但是，这 12 个符号是 Python 的正则表达式中最常用的，可以用来查找和收集绝大多数数据类型。

表2-1：正则表达式常用符号

<table>
<tr><th>符　号</th><th>含　义</th><th>例　子</th><th>匹配结果</th></tr>
<tr><td>*</td><td>匹配前面的字符、子表达式或括号里的字符 0 次或多次</td><td>a*b*</td><td>aaaaaaaa，aaabbbbb，bbbbbb</td></tr>
<tr><td>+</td><td>匹配前面的字符、子表达式或括号里的字符至少 1 次</td><td>a+b+</td><td>aaaaaaab，aaabbbbb，abbbbbb</td></tr>
<tr><td>[]</td><td>匹配任意一个字符（相当于“任选一个”）</td><td>[A-Z]*</td><td>APPLE，CAPITALS，QWERTY</td></tr>
<tr><td>()</td><td>表达式编组（在正则表达式的规则里编组会优先运行）</td><td>(a*b)*</td><td>aaabaab，abaaab，ababaaaaab</td></tr>
<tr><td>{m,n}</td><td>匹配前面的字符、子表达式或括号里的字符 m 到 n 次（包含 m 或 n）</td><td>a{2,3}b{2,3}</td><td>aabbb，aaabbb，aabb</td></tr>
<tr><td>[^]</td><td>匹配任意一个不在中括号里的字符</td><td>[^A-Z]*</td><td>apple，lowercase，qwerty</td></tr>
<tr><td>|</td><td>匹配任意一个由竖线分割的字符、子表达式（注意是竖线，不是大字字母 I）</td><td>b(a|i|e)d</td><td>bad，bid，bed</td></tr>
<tr><td>.</td><td>匹配任意单个字符（包括符号、数字和空格等）</td><td>b.d</td><td>bad，bzd，b$d，b d</td></tr>
<tr><td>^</td><td>指字符串开始位置的字符或子表达式</td><td>^a</td><td>apple，asdf，a</td></tr>
<tr><td>\</td><td>转义字符（把有特殊含义的字符转换成字面形式）</td><td>\.\ | \\</td><td>.|\</td></tr>
<tr><td>$</td><td>经常用在正则表达式的末尾，表示“从字符串的末端匹配”。如果不用它，每个正则表达式实际都带着“.*”模式，只会从字符串开头进行匹配。这个符号可以看成是 ^ 符号的反义词</td><td>[A-Z]*[a-z]*$</td><td>ABCabc，zzzyx，Bob</td></tr>
<tr><td>?!</td><td>“不包含”。这个奇怪的组合通常放在字符或正则表达式前面，表示字符不能出现在目标字符串里。这个符号比较难用，字符通常会在字符串的不同部位出现。如果要在整个字符串中全部排除某个字符，就加上 ^ 和 $ 符号</td><td>^((?![A-Z]).)*$</td><td>no-caps-here，$ymb0ls a4e f!ne</td></tr>
</table>



正则表达式：并非处处正则！

正则表达式的标准版（本书使用的版本，用于 Python 和 BeautifulSoup）是基于 Perl 语法演变而来的。绝大多数主流编程语言都使用与之相同或近似的版本。但是，在其他语言中使用这些正则表达式时需要当心，否则可能会出问题。有些语言，比如 Java，其正则表达式就和 Python 不太一样。总之，遇到问题时看文档！

2.4　正则表达式和BeautifulSoup

如果你觉得前面介绍的正则表达式内容与本书的主题有点儿脱节，那么这里就把它们连接起来。在抓取网页的时候，BeautifulSoup 和正则表达式总是配合使用的。其实，大多数支持字符串参数的函数（比如，`find(id="aTagIdHere")`）都可以用正则表达式实现。

让我们看几个例子，待抓取的网页是 http://www.pythonscraping.com/pages/page3.html。

注意观察网页上有几个商品图片——它们的源代码形式如下：

```
<img src="../img/gifts/img3.jpg">
```

如果我们想抓取所有图片的 URL 链接，非常直接的做法就是用 `findAll("img")` 抓取所有图片，对吗？但是，有个问题。除了那些明显“多余的”图片（比如，LOGO）之外，新式的网站里都有一些隐藏图片，用于网页布局留白和元素对齐的空白图片，以及一些不容易察觉到的图片标签。总之，你不能仅用商品图片来统计网页上所有的图片。

而且网页的布局也可能会变化，或者，因为某些原因，我们不想通过图片在网页中的位置来查找标签。那么当你想抓取随机分布在网站里的某个元素或数据时，就会出现问题。例如，一些网页的最上面可能有一张商品图片，但是在另一些网页上没有。

解决这类问题的办法，就是直接定位那些标签来查找信息。在本例中，我们直接通过商品图片的文件路径来查找：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re

html = urlopen("http://www.pythonscraping.com/pages/page3.html")
bsObj = BeautifulSoup(html)
images = bsObj.findAll("img",{"src":re.compile("\.\.\/img\/gifts/img.*\.jpg")})
for image in images:
    print(image["src"])
```

这段代码会打印出图片的相对路径，都是以 ../img/gifts/img 开头，以 .jpg 结尾，其结果如下所示：

```
../img/gifts/img1.jpg
../img/gifts/img2.jpg
../img/gifts/img3.jpg
../img/gifts/img4.jpg
../img/gifts/img6.jpg
```

正则表达式可以作为 BeautifulSoup 语句的任意一个参数，让你的目标元素查找工作极具灵活性。

2.5　获取属性

到目前为止，我们已经介绍过如何获取和过滤标签，以及获取标签里的内容。但是，在网络数据采集时你经常不需要查找标签的内容，而是需要查找标签属性。比如标签 `<a>` 指向的 URL 链接包含在 `href` 属性中，或者 `<img>` 标签的图片文件包含在 `src` 属性中，这时获取标签属性就变得非常有用了。

对于一个标签对象，可以用下面的代码获取它的全部属性：

```
myTag.attrs
```

要注意这行代码返回的是一个 Python 字典对象，可以获取和操作这些属性。比如要获取图片的资源位置 `src`，可以用下面这行代码：

```
myImgTag.attrs["src"]
```

2.6　Lambda表达式

如果在学校读的是计算机科学专业，那么可能学过 Lambda 表达式，不过可能从来没有用过它。如果你不是计算机科学专业，它们看着可能有点儿陌生（或者只是"曾经学习过的东西"）。在这一节里，虽然我们不打算深入学习这个相当实用的函数，但是会用几个例子来演示它们是如何用在网络数据采集中的。

Lambda 表达式本质上就是一个函数，可以作为其他函数的变量使用；也就是说，一个函数不是定义成 `f(x, y)`，而是定义成 `f(g(x), y)`，或 `f(g(x), h(x))` 的形式。

BeautifulSoup 允许我们把特定函数类型当作 `findAll` 函数的参数。唯一的限制条件是这些函数必须把一个标签作为参数且返回结果是布尔类型。BeautifulSoup 用这个函数来评估它遇到的每个标签对象，最后把评估结果为"真"的标签保留，把其他标签剔除。

例如，下面的代码就是获取有两个属性的标签：

```
soup.findAll(lambda tag: len(tag.attrs) == 2)
```

这行代码会找出下面的标签：

```
<div class="body" id="content"></div>
<span style="color:red" class="title"></span>
```

如果你愿意多写一点儿代码，那么在 BeautifulSoup 里用 Lambda 表达式选择标签，将是正则表达式的完美替代方案。

2.7　超越BeautifulSoup

虽然本书全部用 BeautifulSoup（也是 Python 里最受欢迎的 HTML 解析库之一），但它并不是你唯一的选择。如果 BeautifulSoup 不能满足你的需求，你可以看看其他的库。

- lxml

  这个库（http://lxml.de/）可以用来解析 HTML 和 XML 文档，以非常底层的实现而闻名于世，大部分源代码是用 C 语言写的。虽然学习它需要花一些时间（其实学习曲线越陡峭，表明你可以越快地学会它），但它在处理绝大多数 HTML 文档时速度都非常快。

- HTML parser

  这是 Python 自带的解析库（https://docs.python.org/3/library/html.parser.html）。因为它不用安装（只要装了 Python 就有），所以可以很方便地使用。

第 3 章

开始采集

到目前为止，本书的例子都只是处理单个静态页面，只能算是人为简化的例子（使用作者的网站页面）。从本章开始，我们会看到一些现实问题，需要用爬虫遍历多个页面甚至多个网站。

之所以叫网络爬虫（Web crawler）是因为它们可以沿着网络爬行。它们的本质就是一种递归方式。为了找到 URL 链接，它们必须首先获取网页内容，检查这个页面的内容，再寻找另一个 URL，然后获取 URL 对应的网页内容，不断循环这一过程。

不过要注意的是：你可以这样重复采集网页，但并不意味着你一直都应该这么做。当你需要的所有数据都在一个页面上时，前面例子中的爬虫就足以解决问题了。使用网络爬虫的时候，你必须非常谨慎地考虑需要消耗多少网络流量，还要尽力思考能不能让采集目标的服务器负载更低一些。

3.1　遍历单个域名

即使你没听说过“维基百科六度分隔理论”，也很可能听过“凯文 · 贝肯（Kevin Bacon）的六度分隔值游戏”。在这两个游戏中，都是把两个不相干的主题（维基百科里是用词条之间的连接，凯文 · 贝肯的六度分隔值游戏是用出现在同一部电影中的演员来连接）用一个总数不超过六条的主题连接起来（包括原来的两个主题）。

比如，埃里克 · 艾德尔和布兰登 · 弗雷泽都出现在电影《骑警杜德雷》里，布兰登 · 弗雷泽又和凯文 · 贝肯都出现在电影《我呼吸的空气》里。[1] 因此，根据这两个条件，从埃

注 1：感谢 The Oracle of Bacon（http://oracleofbacon.org/index.php）的存在，满足了我对这类关系链的好奇心。

里克 · 艾德尔到凯文 · 贝肯的链条主题长度只有 3。

我们将在本节创建一个项目来实现"维基百科六度分隔理论"的查找方法。也就是说，我们要实现从埃里克 · 艾德尔的词条页面（https://en.wikipedia.org/wiki/Eric_Idle）开始，经过最少的链接点击次数找到凯文 · 贝肯的词条页面（https://en.wikipedia.org/wiki/Kevin_Bacon）。

这么做对维基百科的服务器负载有多大影响？

根据维基媒体基金会（维基百科归属的组织）的统计，网站每秒钟会收到大约 2500 次点击，其中超过 99% 的点击都是指向维基百科域名（详情请见"维基媒体统计图"（Wikimedia in Figures）里的"流量数据"（Traffic Volume）部分内容，https://meta.wikimedia.org/wiki/Wikimedia_in_figures_-_Wikipedia#Traffic_volume）。因为网站流量很大，所以你的网络爬虫不可能对维基百科的负载有显著影响。不过，如果你频繁地运行本书的代码，或者自己在做项目采集维基百科的词条，那么希望你能够向维基媒体基金会提供一点捐赠（https://wikimediafoundation.org/wiki/Ways_to_Give）——即使是很少的钱来补偿你占用的服务器资源，算是帮助维基百科这个教育资源供其他人使用。

你应该已经知道如何写一段获取维基百科网站的任何页面并提取页面链接的 Python 代码了：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen("http://en.wikipedia.org/wiki/Kevin_Bacon")
bsObj = BeautifulSoup(html)
for link in bsObj.findAll("a"):
    if 'href' in link.attrs:
        print(link.attrs['href'])
```

如果你观察生成的一列链接，就会看到你想要的所有词条链接都在里面："Apollo 13""Philadelphia"和"Primetime Emmy Award"，等等。但是，也有一些我们不需要的链接：

```
//wikimediafoundation.org/wiki/Privacy_policy
//en.wikipedia.org/wiki/Wikipedia:Contact_us
```

其实维基百科的每个页面都充满了侧边栏、页眉、页脚链接，以及连接到分类页面、对话页面和其他不包含词条的页面的链接：

```
/wiki/Category:Articles_with_unsourced_statements_from_April_2014
/wiki/Talk:Kevin_Bacon
```

最近我有个朋友在做一个类似维基百科采集这样的项目，他说为了判断维基百科的内链是否链接到一个词条，他写了一个很大的过滤函数，超过 100 行代码。不幸的是，可能在项目启动的时候，他没有花时间去比较"词条链接"和"其他链接"的差异，也可能他后来发现了那个技巧。如果你仔细观察那些指向词条页面（不是指向其他内容页面）的链接，

会发现它们都有三个共同点：

- 它们都在 id 是 bodyContent 的 div 标签里
- URL 链接不包含分号
- URL 链接都以 /wiki/ 开头

我们可以利用这些规则稍微调整一下代码来获取词条链接：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re

html = urlopen("http://en.wikipedia.org/wiki/Kevin_Bacon")
bsObj = BeautifulSoup(html)
for link in bsObj.find("div", {"id":"bodyContent"}).findAll("a",
                      href=re.compile("^(/wiki/)((?!:).)*$")):
    if 'href' in link.attrs:
        print(link.attrs['href'])
```

如果你运行代码，就会看到维基百科上凯文・贝肯词条里所有指向其他词条的链接。

当然，写程序来找出这个静态的维基百科词条里所有的词条链接很有趣，不过没什么实际用处。我们需要让这段程序更像下面的形式。

- 一个函数 getLinks，可以用维基百科词条 /wiki/< 词条名称 > 形式的 URL 链接作为参数，然后以同样的形式返回一个列表，里面包含所有的词条 URL 链接。
- 一个主函数，以某个起始词条为参数调用 getLinks，再从返回的 URL 列表里随机选择一个词条链接，再调用 getLinks，直到我们主动停止，或者在新的页面上没有词条链接了，程序才停止运行。

完整的代码如下所示：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import datetime
import random
import re

random.seed(datetime.datetime.now())
def getLinks(articleUrl):
    html = urlopen("http://en.wikipedia.org"+articleUrl)
    bsObj = BeautifulSoup(html)
    return bsObj.find("div", {"id":"bodyContent"}).findAll("a",
                     href=re.compile("^(/wiki/)((?!:).)*$"))
links = getLinks("/wiki/Kevin_Bacon")
while len(links) > 0:
    newArticle = links[random.randint(0, len(links)-1)].attrs["href"]
    print(newArticle)
    links = getLinks(newArticle)
```

导入需要的 Python 库之后，程序首先做的是用系统当前时间生成一个随机数生成器。这样可以保证在每次程序运行的时候，维基百科词条的选择都是一个全新的随机路径。

> **伪随机数和随机数种子**
>
> 在前面的示例中，为了能够连续地随机遍历维基百科，我用 Python 的随机数生成器来随机选择每一页上的一个词条链接。但是，用随机数的时候需要格外小心。
>
> 虽然计算机很擅长做精确计算，但是它们处理随机事件时非常不靠谱。因此，随机数是一个难题。大多数随机数算法都努力创造一种呈均匀分布且难以预测的数据序列，但是在算法初始化阶段都需要提供随机数“种子”（random seed）。而完全相同的种子每次将产生同样的“随机”数序列，因此我用系统时间作为随机数序列生成的起点。这样做会让程序运行的时候更具有随机性。
>
> 其实，Python 的伪随机数（pseudorandom number）生成器用的是梅森旋转（Mersenne Twister）算法（https://en.wikipedia.org/wiki/Mersenne_Twister），它产生的随机数很难预测且呈均匀分布，就是有点儿耗费 CPU 资源。真正好的随机数可不便宜！

然后，我们定义 `getLinks` 函数，其参数是维基百科词条页面中 `/wiki/<` 词条名称 `>` 形式的 URL 链接，前面加上维基百科的域名，`http://en.wikipedia.org`，再用域名中的网页获得一个 BeautifulSoup 对象。之后用前面介绍过的参数抽取一列词条链接所在的标签 `a` 并返回它们。

程序的主函数首先把起始页面 https://en.wikipedia.org/wiki/Kevin_Bacon 里的词条链接列表（`links` 变量）设置成链接列表。然后用一个循环，从页面中随机找一个词条链接标签并抽取 `href` 属性，打印这个页面链接，再把这个链接传入 `getLinks` 函数，重新获取新的链接列表。

当然，这里只是简单地构建一个从一个页面到另一个页面的爬虫，要解决“维基百科六度分隔理论”问题还有一点儿工作得做。我们还应该存储 URL 链接数据并分析数据。关于这个问题后续的解决办法，请参考第 5 章内容。

> **异常处理**
>
> 虽然为了方便起见，我们在这些示例中忽略了大多数异常处理过程，但是要注意问题随时可能发生。例如，维基百科改变了 `bodyContent` 标签的名称怎么办呢？（提示：那时代码就会崩溃。）
>
> 因此，这些脚本作为容易演示的示例也许可以运行得很不错，但是要真正成为自动化产品代码，还需要增加更多的异常处理。关于异常处理的更多信息，请参考第 1 章的相关内容。

3.2 采集整个网站

在上一节内容里，我们实现了在一个网站上随机地从一个链接跳到另一个链接。但是，如果你需要系统地把整个网站按目录分类，或者要搜索网站上的每一个页面，怎么办？那就得采集整个网站，那是一种非常耗费内存资源的过程，尤其是处理大型网站时，最合适的工具就是用一个数据库来储存采集的资源。但是，我们可以掌握这类工具的行为，并不需要通过大规模地运行它们。要了解更多关于数据库使用的相关知识，请参考第 5 章。

深网和暗网

你可能听说过**深网**（deep Web）、**暗网**（dark Web）或**隐藏网络**（hidden Web）之类的术语，尤其是在最近的媒体中。它们是什么意思呢？

深网是网络的一部分，与**浅网**（surface Web）对立。浅网是互联网上搜索引擎可以抓到的那部分网络。据不完全统计，互联网中其实约 90% 的网络都是深网。因为谷歌不能做像表单提交这类事情，也找不到那些没有直接链接到顶层域名上的网页，或者因为有 robots.txt 禁止而不能查看网站，所以浅网的数量相对深网还是比较少的。

暗网，也被称为 Darknet 或 dark Internet，完全是另一种"怪兽"。它们也建立在已有的网络基础上，但是使用 Tor 客户端，带有运行在 HTTP 之上的新协议，提供了一个信息交换的安全隧道。这类暗网页面也是可以采集的，就像你采集其他网站一样，不过这些内容超出了本书的范围。

和暗网不同，深网是相对容易采集的。实际上，本书的很多工具都是在教你如何采集那些 Google 爬虫机器人不能获取的深网信息。

那么，什么时候采集整个网站是有用的，而什么时候采集整个网站又是有害无益的呢？遍历整个网站的网络数据采集有许多好处。

- 生成网站地图

  几年前，我曾经遇到过一个问题：一个重要的客户想对一个网站的重新设计方案进行效果评估，但是不想让我们公司进入他们的网站内容管理系统（CMS），也没有一个公开可用的网站地图。我就用爬虫采集了整个网站，收集了所有的链接，再把所有的页面整理成他们网站实际的形式。这让我很快找出了网站上以前不曾留意的部分，并准确地计算出需要重新设计多少网页，以及可能需要移动多少内容。

- 收集数据

  我的另一个客户为了创建一个专业垂直领域的搜索平台，想收集一些文章（故事、博文、新闻等）。虽然这些网站采集并不费劲，但是它们需要爬虫有足够的深度（我们有意收集数据的网站不多）。于是我就创建了一个爬虫递归地遍历每个网站，只收集那些网站页面上的数据。

一个常用的费时的网站采集方法就是从顶级页面开始（比如主页），然后搜索页面上的所有链接，形成列表。再去采集这些链接的每一个页面，然后把在每个页面上找到的链接形成新的列表，重复执行下一轮采集。

很明显，这是一个复杂度增长很快的情形。假如每个页面有 10 个链接，网站上有 5 个页面深度（一个中等规模网站的主流深度），那么如果你要采集整个网站，一共得采集的网页数量就是 105，即 100 000 个页面。不过，虽然“5 个页面深度，每页 10 个链接”是网站的主流配置，但其实很少有网站真的有 100 000 甚至更多的页面。这是因为很大一部分内链都是重复的。

为了避免一个页面被采集两次，链接去重是非常重要的。在代码运行时，把已发现的所有链接都放到一起，并保存在方便查询的列表里（下文示例指 Python 的集合 set 类型）。只有“新”链接才会被采集，之后再从页面中搜索其他链接：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re

pages = set()
def getLinks(pageUrl):
    global pages
    html = urlopen("http://en.wikipedia.org"+pageUrl)
    bsObj = BeautifulSoup(html)
    for link in bsObj.findAll("a", href=re.compile("^(/wiki/)")):
        if 'href' in link.attrs:
            if link.attrs['href'] not in pages:
                # 我们遇到了新页面
                newPage = link.attrs['href']
                print(newPage)
                pages.add(newPage)
                getLinks(newPage)

getLinks("")
```

为了全面地展示这个网络数据采集示例是如何工作的，我降低了在前面例子里使用的“只寻找内链”的标准。不再限制爬虫采集的页面范围，只要遇到页面就查找所有以 /wiki/ 开头的链接，也不考虑链接是不是包含分号。（提示：词条链接不包含分号，而文档上传页面、讨论页面之类的页面 URL 链接都包含分号。）

一开始，用 getLinks 处理一个空 URL，其实是维基百科的主页，因为在函数里空 URL 就是 http://en.wikipedia.org。然后，遍历首页上每个链接，并检查是否已经在全局变量集合 pages 里面了（已经采集的页面集合）。如果不在，就打印到屏幕上，并把链接加入 pages 集合，再用 getLinks 递归地处理这个链接。

关于递归的警告

这个警告在软件开发书籍里很少提到，但是我觉得你应该注意：如果递归运行的次数非常多，前面的递归程序就很可能崩溃。

Python 默认的递归限制（程序递归地自我调用次数）是 1000 次。因为维基百科的网络链接浩如烟海，所以这个程序达到递归限制后就会停止，除非你设置一个较大的递归计数器，或用其他手段不让它停止。

对于那些链接深度少于 1000 的“普通”网站，这个方法通常可以正常运行，一些奇怪的异常除外。例如，我曾经遇到过一个网站，有一个在生成博文内链的规则。这个规则是“当前页面把 /blog/title_of_blog.php 加到它后面，作为本页面的 URL 链接”。

问题是它们可能会把 /blog/title_of_blog.php 加到一个已经有 /blog/ 的 URL 上面了。因此，网站就多了一个 /blog/。最后，我的爬虫找到了这样的 URL 链接：/blog/blog/blog/blog.../blog/title_of_blog.php。

后来，我增加了一些条件，对可能导致无限循环的部分进行检查，确保那些 URL 不是这么荒谬。但是，如果你不去检查这些问题，爬虫很快就会崩溃。

收集整个网站数据

当然，如果只是从一个页面跳到另一个页面，那么网络爬虫是非常无聊的。为了有效地使用它们，在用爬虫的时候我们需要在页面上做些事情。让我们看看如何创建一个爬虫来收集页面标题、正文的第一个段落，以及编辑页面的链接（如果有的话）这些信息。

和往常一样，决定如何做好这些事情的第一步就是先观察网站上的一些页面，然后拟定一个采集模式。通过观察几个维基百科页面，包括词条和非词条页面，比如隐私策略之类的页面，就会得出下面的规则。

- 所有的标题（所有页面上，不论是词条页面、编辑历史页面还是其他页面）都是在 `h1` → `span` 标签里，而且页面上只有一个 `h1` 标签。
- 前面提到过，所有的正文文字都在 `div#bodyContent` 标签里。但是，如果我们想更进一步获取第一段文字，可能用 `div#mw-content-text` → `p` 更好（只选择第一段的标签）。这个规则对所有页面都适用，除了文件页面（例如，https://en.wikipedia.org/wiki/File:Orbit_of_274301_Wikipedia.svg），页面不包含内容文字（`content text`）的部分内容。
- 编辑链接只出现在词条页面上。如果有编辑链接，都位于 `li#ca-edit` 标签的 `li#ca-edit` → `span` → `a` 里面。

调整前面的代码，我们就可以建立一个爬虫和数据收集（至少是数据打印）的组合程序：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re

pages = set()
def getLinks(pageUrl):
   global pages
    html = urlopen("http://en.wikipedia.org"+pageUrl)
    bsObj = BeautifulSoup(html)
    try:
        print(bsObj.h1.get_text())
        print(bsObj.find(id="mw-content-text").findAll("p")[0])
        print(bsObj.find(id="ca-edit").find("span").find("a").attrs['href'])
    except AttributeError:
        print("页面缺少一些属性！不过不用担心！")

    for link in bsObj.findAll("a", href=re.compile("^(/wiki/)")):
        if 'href' in link.attrs:
            if link.attrs['href'] not in pages:
                # 我们遇到了新页面
                newPage = link.attrs['href']
                print("----------------\n"+newPage)
                pages.add(newPage)
                getLinks(newPage)
getLinks("")
```

这个 for 循环和原来的采集程序基本上是一样的（除了打印一条虚线来分离不同的页面内容之外）。

因为我们不可能确保每一页上都有所有类型的数据，所以每个打印语句都是按照数据在页面上出现的可能性从高到低排列的。也就是说，<h1> 标题标签会出现在每一页上（只要能识别，无论哪一页都有），所以我们首先试着获取它的数据。正文内容会出现在大多数页面上（除了文件页面），因此是第二个获取的数据。“编辑”按钮只出现在标题和正文内容都已经获取的页面上，但不是所有这类页面上都有，所以我们最后打印这类数据。

不同模式应对不同需求

在一个异常处理语句中包裹多行语句显然是有点儿危险的。首先，你没法儿识别出究竟是哪行代码出现了异常。其次，如果有个页面没有前面的标题内容，却有“编辑”按钮，那么由于前面已经发生异常，后面的“编辑”按钮链接就不会出现。但是，这种按照网站上信息出现的可能性高低进行排序的方法对许多网站都是可行的，偶而会丢失一点儿数据，只要保存详细的日志就不是什么问题了。

你可能还发现在到目前为止所有的例子中，我们都没有“收集”那些“打印”出来的数据。显然，命令行里显示的数据是很难进一步处理的。我们将在第 5 章继续介绍信息储存和数据库创建的内容。

3.3 通过互联网采集

每次在我做网络数据采集的演讲时，总有人故意问我："你怎么建一个谷歌网站？"我的回答通常会包含两点："首先，你得有几十亿美元能够买得起世界上最大的数据仓库，并把它们隐秘地放在世界各地。其次，你得写一个网络爬虫。"

谷歌在 1994 年成立的时候，就是两个斯坦福大学的毕业生用一个陈旧的服务器和一个 Python 网络爬虫。现在你应该知道了，你已经正式拥有了成为下一个科技亿万富翁需要的工具了！

说句实在话，网络爬虫位于许多新式的网络技术领域彼此交叉的中心地带，而且你使用它们也不需要一个大型数据仓库。要实现任何跨站的数据分析，你只要构建出可以从互联网上无数的网页里解析和储存数据的爬虫就可以了。

就像之前的例子一样，我们后面要建立的网络爬虫也是顺着链接从一个页面跳到另一个页面，描绘出一张网络地图。但是这一次，它们不再忽略外链，而是跟着外链跳转。我们想看看爬虫是不是可以记录我们浏览过的每一个页面上的信息，这将是一个新的挑战。相比我们之前做的单个域名采集，互联网采集要难得多——不同网站的布局迥然不同。这就意味着我们必须在要寻找的信息以及查找方式上都极具灵活性。

不知前方水深浅

下一节的代码可以到达互联网的任何位置。如果我们已经掌握了解决"维基百科六度分隔理论"的方法，那么完全有可能从一个像芝麻街 http://www.sesamestreet.org/ 那样的网站，经过几跳就到达一些非主流网站。

如果读者是小朋友，请在运作代码前咨询一下爸妈。对那些带有敏感题材或有宗教限制的人来说，某些网站是禁止浏览的，阅读代码示例没问题，但是运行代码的时候请格外小心。

在你写爬虫随意跟随外链跳转之前，请问自己几个问题。

- 我要收集哪些数据？这些数据可以通过采集几个已经确定的网站（永远是最简单的做法）完成吗？或者我的爬虫需要发现那些我可能不知道的网站吗？
- 当我的爬虫到了某个网站，它是立即顺着下一个出站链接跳到一个新网站，还是在网站上呆一会儿，深入采集网站的内容？
- 有没有我不想采集的一类网站？我对非英文网站的内容感兴趣吗？
- 如果我的网络爬虫引起了某个网站网管的怀疑，我如何避免法律责任？（关于这个问题的更多信息请参考附录 C。）

几个灵活的 Python 函数组合起来就可以实现不同类型的网络爬虫，用不超过 50 行代码就可轻松地写出来：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
import datetime
import random

pages = set()
random.seed(datetime.datetime.now())

# 获取页面所有内链的列表
def getInternalLinks(bsObj, includeUrl):
    internalLinks = []
    # 找出所有以"/"开头的链接
    for link in bsObj.findAll("a", href=re.compile("^(/|.*"+includeUrl+")")):
        if link.attrs['href'] is not None:
            if link.attrs['href'] not in internalLinks:
                internalLinks.append(link.attrs['href'])
    return internalLinks

# 获取页面所有外链的列表
def getExternalLinks(bsObj, excludeUrl):
    externalLinks = []
    # 找出所有以"http"或"www"开头且不包含当前URL的链接
    for link in bsObj.findAll("a",
                         href=re.compile("^(http|www)((?!"+excludeUrl+").)*$")):
        if link.attrs['href'] is not None:
            if link.attrs['href'] not in externalLinks:
                externalLinks.append(link.attrs['href'])
    return externalLinks

def splitAddress(address):
    addressParts = address.replace("http://", "").split("/")
    return addressParts

def getRandomExternalLink(startingPage):
    html = urlopen(startingPage)
    bsObj = BeautifulSoup(html)
    externalLinks = getExternalLinks(bsObj, splitAddress(startingPage)[0])
    if len(externalLinks) == 0:
        internalLinks = getInternalLinks(startingPage)
        return getNextExternalLink(internalLinks[random.randint(0,
                                  len(internalLinks)-1)])
    else:
        return externalLinks[random.randint(0, len(externalLinks)-1)]

def followExternalOnly(startingSite):
    externalLink = getRandomExternalLink("http://oreilly.com")
    print("随机外链是:"+externalLink)
    followExternalOnly(externalLink)

followExternalOnly("http://oreilly.com")
```

上面这个程序从 http://oreilly.com 开始，然后随机地从一个外链跳到另一个外链。输出的结果如下所示：

```
随机外链是:http://igniteshow.com/
随机外链是:http://feeds.feedburner.com/oreilly/news
随机外链是:http://hire.jobvite.com/CompanyJobs/Careers.aspx?c=q319
随机外链是:http://makerfaire.com/
```

网站首页上并不能保证一直能发现外链。这时为了能够发现外链，就需要用一种类似前面案例中使用的采集方法，即递归地深入一个网站直到找到一个外链才停止。

图 3-1 把程序操作可视化成了一个流程图。

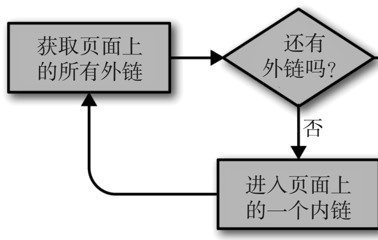


图 3-1：从互联网上不同的网站采集外链的程序流程图



不要把示例程序放进产品代码

我想把代码写得更完整，但是写书的时候空间和可读性非常重要，所以书中的示例程序没有包含真实产品代码中必须有的检查和异常处理。

例如，如果爬虫遇到一个网站里面一个外链都没有（虽然不太可能，但是如果程序运行的时候够长总会遇到这类情况），这时程序就会一直在这个网站运行跳不出去，直到递归到达 Python 的限制为止。

在以任何正式的目的运行代码之前，请确认你已经在可能出现问题的地方都放置了检查语句。

把任务分解成像“获取页面上所有外链”这样的小函数是不错的做法，以后可以方便地修改代码以满足另一个采集任务的需求。例如，如果我们的目标是采集一个网站所有的外链，并且记录每一个外链，我们可以增加下面的函数：

```
# 收集网站上发现的所有外链列表
allExtLinks = set()
allIntLinks = set()
def getAllExternalLinks(siteUrl):
```

```
        html = urlopen(siteUrl)
        bsObj = BeautifulSoup(html)
        internalLinks = getInternalLinks(bsObj,splitAddress(siteUrl)[0])
        externalLinks = getExternalLinks(bsObj,splitAddress(siteUrl)[0])
        for link in externalLinks:
            if link not in allExtLinks:
                allExtLinks.add(link)
                print(link)
        for link in internalLinks:
            if link not in allIntLinks:
                print("即将获取链接的URL是:"+link)
                allIntLinks.add(link)
                getAllExternalLinks(link)

    getAllExternalLinks("http://oreilly.com")
```

这段代码可以看出两个循环——一个是收集内链，一个是收集外链——然后彼此连接起来工作，程序的流程如图 3-2 所示。

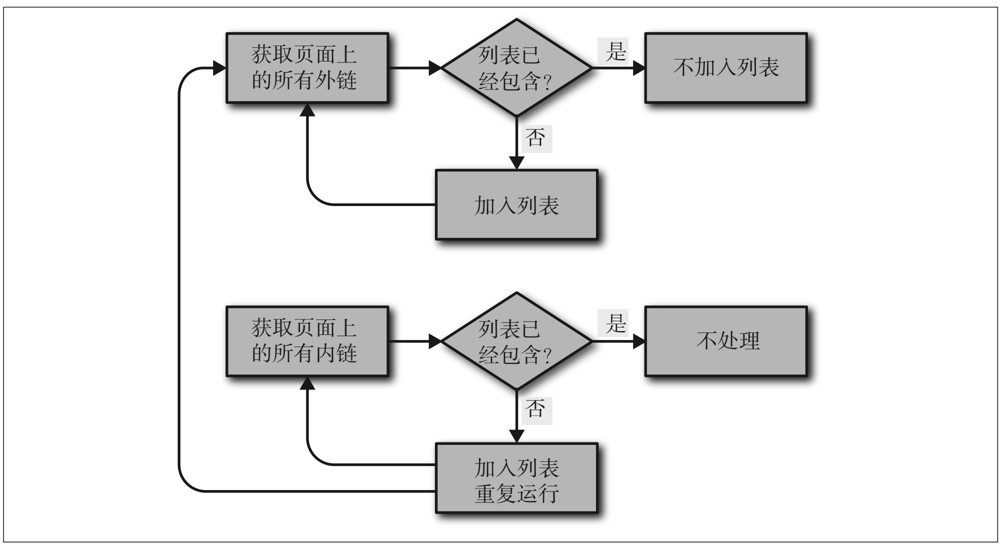


图 3-2：收集内链和外链的程序流程图

写代码之前拟个大纲或画个流程图是很好的编程习惯，这么做不仅可以为你后期处理节省很多时间，更重要的是可以防止自己在爬虫变得越来越复杂时乱了分寸。

处理网页重定向

重定向（redirect）允许一个网页在不同的域名下显示。重定向有两种形式：

- 服务器端重定向，网页在加载之前先改变了 URL；
- 客户端重定向,有时你会在网页上看到“10 秒钟后页面自动跳转到……”之类的消息，表示在跳转到新 URL 之前网页需要加载内容。

本节处理的是服务器端重定向的内容。更多关于客户端重定向的细节，通常用 JavaScript 或 HTML 来实现，请看第 10 章。

服务器端重定向，你通常不用担心。如果你在用 Python 3.x 版本的 `urllib` 库，它会自动处理重定向。不过要注意，有时候你要采集的页面的 URL 可能并不是你当前所在页面的 URL。

3.4 用Scrapy采集

写网络爬虫的挑战之一是你经常需要不断地重复一些简单任务：找出页面上的所有链接，区分内链与外链，跳转到新的页面。掌握这些基本模式非常有用，从零开始编写也完全可行，不过有几个工具可以帮你自动处理这些细节。

Scrapy 就是一个帮你大幅度降低网页链接查找和识别工作复杂度的 Python 库，它可以让你轻松地采集一个或多个域名的信息。不过目前 Scrapy 仅支持 Python 2.7，还不支持 Python 3.x。

当然在一台机器上同时使用多个版本的 Python 是没有问题的（比如，同时安装 Python 2.7 和 Python 3.4）。如果你既想用 Scrapy 做项目，又想用 Python 3.4 写程序，完全没问题。

Scrapy 网站提供了最新版工具的下载页面（http://scrapy.org/download/），也可以用 pip 等第三方安装包安装。记住 Python 的版本必须是 2.7（2.6 和 3.x 都不兼容），而且运行所有使用 Scrapy 的程序也必须在 Python 2.7 环境下。

虽然写 Scrapy 爬虫很简单，但完成一个爬虫还是需要一些设置。如果在当前目录下创建新的 Scrapy 项目，就执行下面的代码：

```
$scrapy startproject wikiSpider
```

`wikiSpider` 是新项目的名称。在当前目录中会新建一个名称也是 wikiSpider 的项目文件夹。文件夹的目录结构如下所示：

- scrapy.cfg
  — wikiSpider

— __init.py__

— items.py

— pipelines.py

— settings.py

— spiders

— __init.py__

为了创建一个爬虫，我们需要在 wikiSpider/wikiSpider/spiders/ 文件夹里增加一个 articleSpider.py 文件。另外，在 items.py 文件中，我们需要定义一个 `Article` 类。

你的 items.py 文件应该像下面这样（Scrapy 自动生成的注释内容可以保留，当然删除也可以）：

```
# -*- coding: utf-8 -*-
# Define here the models for your scraped items
#
# See documentation in:
# http://doc.scrapy.org/en/latest/topics/items.html

from scrapy import Item, Field

class Article(Item):
    # define the fields for your item here like:
    # name = scrapy.Field()
    title = Field()
```

Scrapy 的每个 `Item`（条目）对象表示网站上的一个页面。当然，你可以根据需要定义不同的条目（比如 `url`、`content`、`header image` 等），但是现在我只演示收集每页的 `title` 字段（field）。

在新建的 articleSpider.py 文件里面，写如下代码：

```
from scrapy.selector import Selector
from scrapy import Spider
from wikiSpider.items import Article


class ArticleSpider(Spider):
    name="article"
    allowed_domains = ["en.wikipedia.org"]
    start_urls = ["http://en.wikipedia.org/wiki/Main_Page",
              "http://en.wikipedia.org/wiki/Python_%28programming_language%29"]

    def parse(self, response):
        item = Article()
        title = response.xpath('//h1/text()')[0].extract()
        print("Title is: "+title)
        item['title'] = title
        return item
```

这个类的名称（ArticleSpider）与爬虫文件的名称（wikiSpider）是不同的，这个类只是在 wikiSpider 目录里的一员，仅仅用于维基词条页面的采集。对一些信息类型较多的大网站，你可能会为每种信息（如博客的博文、图书出版发行信息、专栏文章等）设置独立的 Scrapy 条目，每个条目都有不同的字段，但是所有条目都在同一个 Scrapy 项目里运行。

你可以在 wikiSpider 主目录中用如下命令运行 ArticleSpider：

```
$ scrapy crawl article
```

这行命令会用条目名称 article 来调用爬虫（不是类名，也不是文件名，而是由 ArticleSpider 的 name = "article" 决定的）。

陆续出现的调试信息中应该会这两行结果：

```
Title is: Main Page
Title is: Python (programming language)
```

这个爬虫先进入 start_urls 里面的两个页面，收集信息，然后停止。虽然这个爬虫很简单，但是如果你有许多 URL 需要采集，Scrapy 这种用法会非常适合。为了让爬虫更加完善，你需要定义一些规则让 Scrapy 可以在每个页面查找 URL 链接：

```
from scrapy.contrib.spiders import CrawlSpider, Rule
from wikiSpider.items import Article
from scrapy.contrib.linkextractors.sgml import SgmlLinkExtractor

class ArticleSpider(CrawlSpider):
    name="article"
    allowed_domains = ["en.wikipedia.org"]
    start_urls = ["http://en.wikipedia.org/wiki/Python_
                  %28programming_language%29"]
    rules = [Rule(SgmlLinkExtractor(allow=('(/wiki/)((?!:).)*$'),),
                                   callback="parse_item", follow=True)]

    def parse_item(self, response):
        item = Article()
        title = response.xpath('//h1/text()')[0].extract()
        print("Title is: "+title)
        item['title'] = title
        return item
```

虽然这个爬虫和前面那个爬虫的启动命令一样，但是如果你不用 Ctrl+C 中止程序，它是不会停止的（很长时间也不会停止）。

Scrapy 日志处理

Scrapy 生成的调试信息非常有用，但是通常太罗嗦。你可以在 Scrapy 项目中的 setting.py 文件中设置日志显示层级：

```
LOG_LEVEL = 'ERROR'
```

Scrapy 日志有五种层级，按照范围递增顺序排列如下：

- CRITICAL
- ERROR
- WARNING
- DEBUG
- INFO

如果日志层级设置为 ERROR，那么只有 CRITICAL 和 ERROR 日志会显示出来。如果日志层级设置为 INFO，那么所有信息都会显示出来，其他同理。

日志不仅可以显示在终端，也可以通过下面命令输出到一个独立的文件中：

```
$ scrapy crawl article -s LOG_FILE=wiki.log
```

如果目录中没有 wiki.log，那么运行程序会创建一个新文件，然后把所有的日志都保存到里面。如果已经存在，会在原文后面加入新的日志内容。

Scrapy 用 `Item` 对象决定要从它浏览的页面中提取哪些信息。Scrapy 支持用不同的输出格式来保存这些信息，比如 CSV、JSON 或 XML 文件格式，对应命令如下所示：

```
$ scrapy crawl article -o articles.csv -t csv
$ scrapy crawl article -o articles.json -t json
$ scrapy crawl article -o articles.xml -t xml
```

当然，你也可以自定义 `Item` 对象，把结果写入你需要的一个文件或数据库中，只要在爬虫的 `parse` 部分增加相应的代码即可。

Scrapy 是处理网络数据采集相关问题的利器。它可以自动收集所有 URL，然后和指定的规则进行比较；确保所有的 URL 是唯一的；根据需求对相关的 URL 进行标准化；以及到更深层的页面中递归查找。

尽管这点内容算只能是碰到了 Scrapy 强大功能的一角，但我依然鼓励你去学习 Scrapy 文档（http://doc.scrapy.org/en/latest/）和其他在线的学习资源。Scrapy 的内容非常丰富，具有很多特性。如果那些你想用 Scrapy 做的事情在这里没有提到，那么 Scrapy 很可能有一种（或几种）方法可以满足你的需求。

第4章

使用API

我和许多参与过大项目的程序员一样，在使用别人的代码时，也曾有过惨痛的经历。从命名空间问题到函数输出类型的问题，即使像获取指针 A 的信息然后传递给方法 B 这样简单的事情，也仿佛一场噩梦。

这正是应用编程接口（Application Programming Interface，API）的用处：它们为不同的应用提供了方便友好的接口。不同的开发者用不同的架构，甚至不同的语言编写软件都没问题——因为 API 设计的目的就是要成为一种通用语言，让不同的软件进行信息共享。

尽管目前不同的软件应用都有各自不同的 API，但“API”经常被看成“网络应用 API”。一般情况下，程序员可以用 HTTP 协议向 API 发起请求以获取某种信息，API 会用 XML（eXtensible Markup Language，可扩展标记语言）或 JSON（JavaScript Object Notation，JavaScript 对象表示）格式返回服务器响应的信息。尽管大多数 API 仍然在用 XML，但是 JSON 正在快速成为数据编码格式的主流选择。

用这种即开即用的接口获取预先打包好的信息，看起来好像和本书主题没什么关系，但是这种看法只对了一半。虽然大多数人通常不会把使用 API 看成网络数据采集，但是实际上两者使用的许多技术（都是发送 HTTP 请求）和产生的结果（都是获取信息）差不太多；两者经常是相辅相成的关系。

例如，你可能会把网络爬虫和 API 获取的信息组合起来，因为这样的信息可能更有意义。在本章后面的一个例子中，我们将会介绍如何把维基百科编辑历史（里面有编辑者 IP 地址）和一个 IP 地址解析的 API 组合起来，以获取维基百科词条编辑者的地理位置。

在这一章里，我们将首先概述 API，之后介绍 API 的工作原理，以及几个目前比较流行的 API，最后介绍如何在网络爬虫里使用这些 API。

4.1 API概述

虽然 API 并非随处可见（这正是我写这本书的主要动机，因为即使你找不到 API，也可以用爬虫采集信息），但是你可以从 API 里获取许多信息。如果你对音乐感兴趣，有几个 API 可以为你提供歌曲名称、歌手、专辑，以及歌曲风格和相关歌手的信息。想要体育信息？ ESPN 提供的 API 包括运动员信息、比赛分数等。Google 的开发者社区（https://console.developers.google.com/）也提供了一堆 API 用于获取语言翻译、分析、地理位置等信息。

API 很容易使用。其实你只要在浏览器里输入下面的网址就可以发起一个简单的 API 请求：[1]

http://freegeoip.net/json/50.78.253.58

应该会出现下面的结果：


```
{"ip":"50.78.253.58","country_code":"US","country_name":"美国","region_
code":"MA","region_name":"Massachusetts","city":"Chelmsford","zipcode":"01824",
"latitude":42.5879,"longitude":-71.3498,"metro_code":"506","area_code":"978"}
```


你可能会想，这不就是在浏览器窗口输入一个网址，按回车后获取的（只是 JSON 格式）信息吗？究竟 API 和普通的网址访问有什么区别呢？如果不考虑 API 高大上的名称，其实两者没啥区别。API 可以通过 HTTP 协议下载文件，和 URL 访问网站获取数据的协议一样，它几乎可以实现所有在网上干的事情。API 之所以叫 API 而不是叫网站的原因，其实是首先 API 请求使用非常严谨的语法，其次 API 用 JSON 或 XML 格式表示数据，而不是 HTML 格式。

4.2 API通用规则

和大多数网络数据采集的方式不同，API 用一套非常标准的规则生成数据，而且生成的数据也是按照非常标准的方式组织的。因为规则很标准，所以一些简单、基本的规则很容易学，可以帮你快速地掌握任意 API 的用法。

不过并非所有 API 都很简单，有些 API 的规则比较复杂，因此第一次使用一个 API 时，建议阅读文档，无论你对以前用过的 API 是多么熟悉。

注 1：这个 API 把 IP 地址解析成地理位置，在本章的后面还会用到。你可以通过 http://freegeoip.net/ 看到更多的信息。

4.2.1　方法

利用 HTTP 从网络服务获取信息有四种方式：

- GET
- POST
- PUT
- DELETE

GET 就是你在浏览器中输入网址浏览网站所做的事情。当你访问 http://freegeoip.net/json/50.78.253.58 时，就会使用 GET 方法。可以想象成 GET 在说："喂，网络服务器，请按照这个网址给我信息。"

POST 基本就是当你填写表单或提交信息到网络服务器的后端程序时所做的事情。每次当你登录网站的时候，就是通过用户名和（有可能加密的）密码发起一个 POST 请求。如果你用 API 发起一个 POST 请求，相当于说"请把信息保存到你的数据库里"。

PUT 在网站交互过程中不常用，但是在 API 里面有时会用到。PUT 请求用来更新一个对象或信息。例如，API 可能会要求用 POST 请求创建新用户，但是如果你要更新老用户的邮箱地址，就要用 PUT 请求了。[2]

DELETE 用于删除一个对象。例如，如果我们向 http://myapi.com/user/23 发出一个 DELETE 请求，就会删除 ID 号是 23 的用户。DELETE 方法在公共 API 里面不常用，它们主要用于创建信息，不能随便让一个用户去删掉数据库的信息。但是，和 PUT 方法一样，DELETE 方法也值得了解一下。

虽然在 HTTP 规范里还有一些信息处理方式，但是这四种基本是你使用 API 过程中可能遇到的全部。

4.2.2　验证

虽然有些 API 不需要验证操作（就是说任何人都可以使用 API，不需要注册），但是很多新式 API 在使用之前都要求客户验证。

有些 API 要求客户验证是为了计算 API 调用的费用，或者是提供了包月的服务。有些验证是为了"限制"用户使用 API（限制每秒钟、每小时或每天 API 调用的次数），或者是限制一部分用户对某种信息或某类 API 的访问。还有一些 API 可能不要求验证，但是可能会

注 2：其实，很多 API 在更新信息的时候都是用 POST 请求代替 PUT 请求。究竟是创建一个新实体还是更新一个旧实体，通常要看 API 请求本身是如何构建的。不过，掌握两者的差异还是有好处的，用 API 的时候你经常会遇到 PUT 请求。

为了市场营销而跟踪用户的使用行为。

通常 API 验证的方法都是用类似令牌（token）的方式调用，每次 API 调用都会把令牌传递到服务器上。这种令牌要么是用户注册的时候分配给用户，要么就是在用户调用的时候才提供，可能是长期固定的值，也可能是频繁变化的，通过服务器对用户名和密码的组合处理后生成。

例如，调用 The Echo Nest 音乐平台的 API 获取枪与玫瑰乐队（Guns N'Roses）的歌曲：

```
http://developer.echonest.com/api/v4/artist/songs?api_key=<你的api_key>
%20&name=guns%20n%27%20roses&format=json&start=0&results=100
```

这个链接向服务器提供的 `api_key` 是我注册之后得到的，服务器会识别出这个链接发起的是 Ryan Mitchell（我）的请求，然后向请求者提供 JSON 格式的数据。

令牌除了在 URL 链接中传递，还会通过请求头里的 cookie 把用户信息传递给服务器。我们将在本章后面和第 12 章更加详细地介绍请求头的内容，这里仅做简单的演示，请求头可以用前几章使用的 `urllib` 包进行传递。

```
token = "<your api key>"
webRequest = urllib.request.Request("http://myapi.com", headers={"token":token})
html = urlopen(webRequest)
```

4.3 服务器响应

和本章前面介绍的 FreeGeoIP 的例子一样，API 有一个重要的特征是它们会反馈格式友好的数据。大多数反馈的数据格式都是 XML 和 JSON。

这几年，JSON 比 XML 更受欢迎，主要有两个原因。首先，JSON 文件比完整的 XML 格式小。比如下面的 XML 数据用了 98 个字符：

```
<user><firstname>Ryan</firstname><lastname>Mitchell</lastname><username>Kludgist
</username></user>
```

同样的 JSON 格式数据：

```
{"user":{"firstname":"Ryan","lastname":"Mitchell","username":"Kludgist"}}
```

只要用 73 个字符，比表述同样内容的 XML 文件要小 36%。

当然有人可能会说，XML 也可以表示成这种形式：

```
<user firstname="ryan" lastname="mitchell" username="Kludgist"></user>
```

不过这么做并不好，因为它不支持深层嵌入数据。而且它也用了 71 个字符，和 JSON 差不多。

JSON 格式比 XML 更受欢迎的另一个原因是网络技术的改变。过去，服务器端用 PHP 和 .NET 这些程序作为 API 的接收端。现在，服务器端也会用一些 JavaScript 框架作为 API 的发送和接收端，像 Angular 或 Backbone 等。虽然服务器端的技术无法预测它们即将收到的数据格式，但是像 Backbone 之类的 JavaScript 库处理 JSON 比处理 XML 要更简单。

虽然大多数 API 都支持 XML 数据格式，但在本书中我们还是用 JSON 格式。当然，如果你还没有把两种格式都掌握，那么现在熟悉它们是个好时机——短期内它们都不会消失。

API调用

不同 API 的调用语法大不相同，但是有几条共同准则。当使用 `GET` 请求获取数据时，用 URL 路径描述你要获取的数据范围，查询参数可以作为过滤器或附加请求使用。

例如，下面这个虚拟的 API，可以获取 ID 是 1234 的用户在 2014 年 8 月份发表的所有博文：

```
http://socialmediasite.com/users/1234/posts?from=08012014&to=08312014
```

有许多 API 会通过文件路径（path）的形式指定 API 版本、数据格式和其他属性。例如，下面的链接会返回同样的结果，但是使用虚拟 API 的第四版，反馈数据为 JSON 格式：

```
http://socialmediasite.com/api/v4/json/users/1234/posts?from=08012014&to=08312014
```

还有一些 API 会通过请求参数（request parameter）的形式指定数据格式和 API 版本：

```
http://socialmediasite.com/users/1234/posts?format=json&from=08012014&to=08312014
```

4.4 Echo Nest

The Echo Nest 音乐数据网站[3]是一个用网络爬虫建立的超级给力的企业级案例。虽然像 Pandora 之类的音乐公司都是通过人工干预完成音乐的分类与说明，但是 The Echo Nest 是通过自动智能技术，以及博客与新闻信息的采集，来完成艺术家、歌曲和专辑的分类工作的。

更给力的是，它的 API 可以经非商业用途免费使用。[4]使用 API 得有一个 key，你可以在 The Echo Nest 的注册页面（https://developer.echonest.com/account/register）填入名称、邮箱和用户名来注册账号。

注 3：2005 年成立，2014 年被 Spotify 以 1 亿美元收购。

注 4：请看 The Echo Nest 的授权页面（http://developer.echonest.com/licensing.html）中关于请求限制的相关细节。

几个示例

The Echo Nest 的 API 的响应结果由四个部分组成：艺术家（artist）、歌曲（song）、专辑（track）和风格（genre）。除了风格之外，所有信息都带有唯一的 ID 号，可以通过 API 调用把信息展示成不同的形式。假如我想获取 Monty Python 喜剧乐团的歌曲，可以用下面的链接获取歌曲的 ID（记得把 < 你的 `api_key`> 替换成你自己的 API key）：

```
http://developer.echonest.com/api/v4/artist/search?api_key=<你的api_
 key>&name=monty%20python
```

响应的结果是：

```
{"response": {"status": {"version": "4.2", "code": 0, "message": "Suc
cess"}, "artists": [{"id": "AR5HF791187B9ABAF4", "name": "Monty Pytho
n"}, {"id": "ARWCIDE13925F19A33", "name": "Monty Python's SPAMALOT"},
 {"id": "ARVPRCC12FE0862033", "name": "Monty Python's Graham Chapman"
}]}}
```

还可以用歌曲的 ID 号查询歌曲名称：

```
http://developer.echonest.com/api/v4/artist/songs?api_key=<你的api_key>&id=
AR5HF791187B9ABAF4&format=json&start=0&results=10
```

这样就会响应 Monty Python 的歌曲查询结果，都是一些不太流行的歌曲：

```
{"response": {"status": {"version": "4.2", "code": 0, "message": "Success"},
"start": 0, "total": 476, "songs": [{"id": "SORDAUE12AF72AC547", "title":
"Neville Shunt"}, {"id": "SORBMPW13129A9174D", "title": "Classic (Silbury Hill)
(Part 2)"}, {"id": "SOQXAYQ1316771628E", "title": "Famous Person Quiz (The
Final Rip Off Remix)"}, {"id": "SOUMAYZ133EB4E17E8", "title": "Always Look On
The Bright Side Of Life - Monty Python"}, ...]}}
```

另外，我也可以用 `name` 是 `monty%20python` 来替换唯一的 ID 号来获取同样的信息：

```
http://developer.echonest.com/api/v4/artist/songs?api_key=<你的api_key>2&name=
monty%20python&format=json&start=0&results=10
```

用同样的 ID 号，我也可以请求与 Monty Python 风格相似的艺术家：

```
http://developer.echonest.com/api/v4/artist/similar?api_key=<你的api_key>&id=
AR5HF791187B9ABAF4&format=json&results=10&start=0
```

响应的结果包括像 Eric Idle 那样的喜剧艺术家，他是 Monty Python 的一员：

```
{"response": {"status": {"version": "4.2", "code": 0, "message": "Suc
cess"}, "artists": [{"name": "Life of Brian", "id": "ARNZYOS1272BA7FF
38"}, {"name": "Eric Idle", "id": "ARELDIS1187B9ABC79"}, {"name": "Th
e Simpsons", "id": "ARNR4B91187FB5027C"}, {"name": "Tom Lehrer", "id"
: "ARJMYTZ1187FB54669"}, ...]}}
```

你会发现这些相似艺术家包含一些很有趣的信息（比如，“Tom Lehrer”[5]），第一个结果“The Life of Brian”是 Monty Python 乐团演奏的电影配乐。使用这类取材丰富但人工干预很少的数据库时，比较痛苦的是有时候会遇到一些无厘头的结果。这在使用第三方 API 创建应用时需要格外注意。

我就介绍这几个 The Echo Nest API 的小例子。具体文档请参考 The Echo Nest API 概述（http://developer.echonest.com/docs/v4）。

The Echo Nest 资助了很多技术与音乐交叉领域的黑客松项目（hackathon，也叫黑客马拉松、编程马拉松）和编程项目。如果你想从中获取灵感，The Echo Nest 示例页面（http://static.echonest.com/labs/demo.html）是一个好的起点。

4.5 Twitter API

众所周知，Twitter 非常保护自己的 API，这也是理所当然的。这家公司平均每月拥有 2.3 亿活跃用户和超过 1 亿美元的收入，是不会愿意让用户随意获取信息的。

Twitter 的 API 请求限制有两种方法：每 15 分钟 15 次和每 15 分钟 180 次，由请求类型决定。比如你可以 1 分钟获取 12 次（每 15 分钟 180 次的平均数）Twitter 用户基本信息，但是 1 分钟只能获取 1 次（每 15 分钟 15 次的平均数）这些用户的关注者（follower）。[6]

4.5.1 开始

除了流量限制，Twitter 的 API 验证方式也比 The Echo Nest 要复杂，既要有 API 的 key，也要用其他 key。要获取 API 的 key，你需要注册一个 Twitter 账号；可以在注册页面（https://twitter.com/signup）直接注册。另外，还需要在 Twitter 的开发者网站（https://apps.twitter.com/app/new）注册一个新应用。

完成注册之后，你会在一个新页面看到你应用的基本信息，包括自定义的 key（图 4-1）。

注 5：美国歌手、数学家、曲风简洁幽默，https://en.wikipedia.org/wiki/Tom_Lehrer。——译者注

注 6：完整的流量限制列表，请查看 https://dev.twitter.com/rest/public/rate-limits。

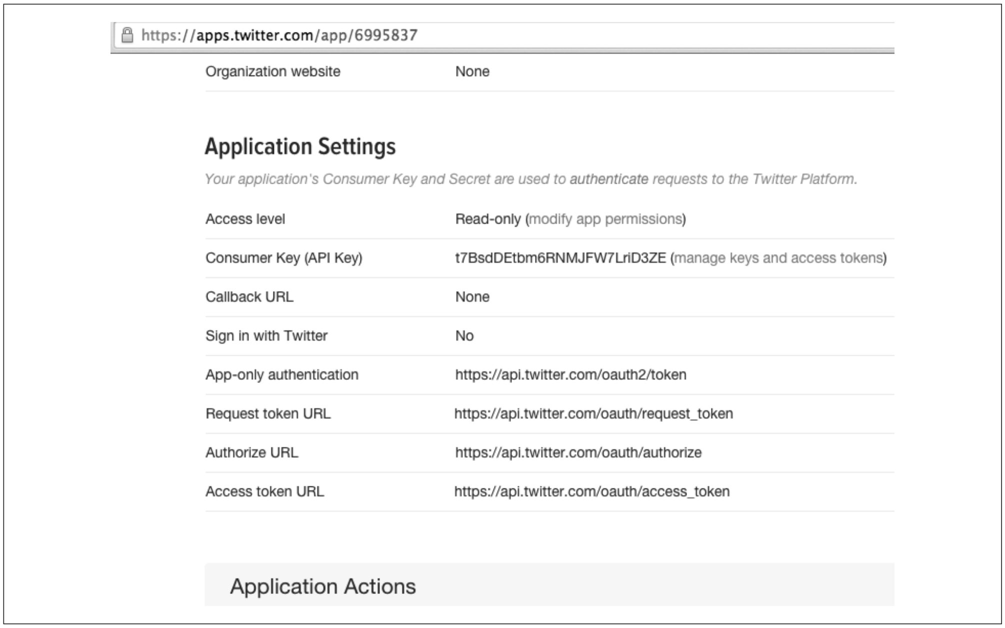


图 4-1：Twitter 的应用设置页面提供了新应用的基本信息

如果你单击“manage keys and access tokens”页面，就会跳转到一个包含更多信息的页面上（图 4-2）。

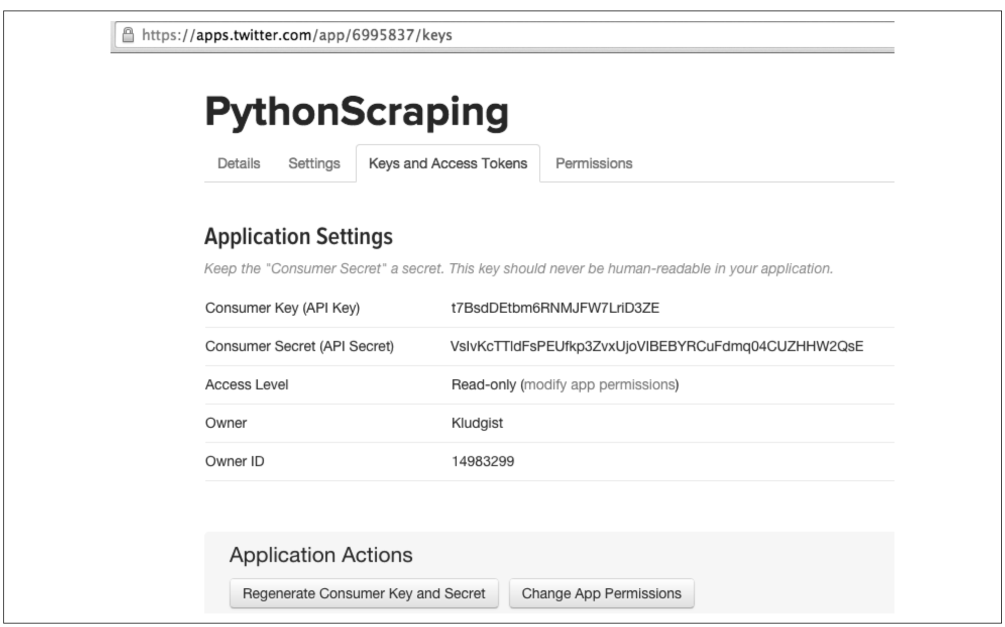


图 4-2：使用 Twitter 的 API 需要用加密 key

这个页面还包括一个自动生成加密 key 的按钮，可以使得应用被公开访问（比如，你打算把这个应用作为一本书里的例子使用时）。

4.5.2　几个示例

Twitter 的验证系统用 OAuth 验证，非常复杂；最好找一个成熟稳定的 Python 库来处理它，不要自己从头写代码来实现。因为手工处理 Twitter 的 API 是非常复杂的工作，所以本节内容的重点是用 Python 代码来实现 API 的交互，不是亲手实现这个 API。

在编写本书时，有很多 Python 2.x 版本的库可以与 Twitter 进行交互，但是 Python 3.x 版本的库比较少。好在最好的一个 Python Twitter 库（名字也叫 Twitter）也支持 Python 3.x 版本。你可以从 Python Twitter Tools（PTT，http://mike.verdone.ca/twitter/#downloads）网站下载并安装这个库（pip 安装也可以，`pip install twitter`）：

```
$cd twitter-x.xx.x
$python setup.py install
```

Twitter 访问权限

应用的默认访问权限（credential permissions）是只读（read-only）模式，除了让你的应用发推文之外，这样的权限可以满足大部分需求。

如果想把令牌的权限改成读 / 写（read/write）模式，你可以在 Twitter 的应用控制面板的权限栏进行修改。改变权限后令牌会重新生成。

如果有需要你也可以更新应用的令牌权限，用它登录你的 Twitter 账号直接收发推文。不过要注意信息安全。通常，应该对不同的应用授予不同的权限，而不是给那些不需要太多权限的应用过多的访问权限。

我们的第一个练习是搜索某个推文。下面的代码连接 Twitter API，然后打印一个包含 `#python` 标签的推文 JSON 列表。记得用的时候把对应的信息替换成你的 OAuth 验证信息：

```
from twitter import Twitter

t = Twitter(auth=OAuth(<Access Token>,<Access Token Secret>,
                       <Consumer Key>,<Consumer Secret>))
pythonTweets = t.search.tweets(q = "#python")
print(pythonTweets)
```

虽然这个程序的打印结果可能会很长，但是你可以获得推文的所有信息，包括：推文的发表日期和具体时间，转发或收藏的信息，用户账号和简介图片的信息，等等。虽然你只想看这些推文的部分内容，但是 Twitter API 是为那些想在自己网站上显示完整推文的开发者设计的，因此会包含许多内容。

你也可以通过 API 发一篇推文来看看效果：

```
from twitter import *

t = Twitter(auth=OAuth(<Access Token>, <Access Token Secret>,
                       <Consumer Key>, <Consumer Secret>))
statusUpdate = t.statuses.update(status='Hello, world!')
print(statusUpdate)
```

推文的 JSON 格式数据（JSON 字段和内容都用双引号，这是 Python 字符串打印形式的内容）如下所示：


```
{'created_at': 'Sun Nov 30 07:23:39 +0000 2014', 'place': None, 'in_reply_to_scr
een_name': None, 'id_str': '538956506478428160', 'in_reply_to_user_id: None,'lan
g': 'en', 'in_reply_to_user_id_str': None, 'user': {'profile_sidebar_border_colo
r': '000000', 'profile_background_image_url': 'http://pbs.twimg.com/profile_back
ground_images/497094351076347904/RXn8MUlD.png', 'description':'Software Engine
er@LinkeDrive, Masters student @HarvardEXT, @OlinCollege graduate, writer @OReil
lyMedia. Really tall. Has pink hair. Female, despite the name.','time_zone': 'Ea
stern Time (US & Canada)', 'location': 'Boston, MA', 'lang': 'en', 'url': 'http:
//t.co/FM6dHXloIw', 'profile_location': None, 'name': 'Ryan Mitchell', 'screen_n
ame': 'Kludgist', 'protected': False, 'default_profile_image': False, 'id_str':
'14983299', 'favourites_count': 140, 'contributors_enabled': False, 'profile_use
_background_image': True, 'profile_background_image_url_https': 'https://pbs.twi
mg.com/profile_background_images/497094351076347904/RXn8MUlD.png', 'profile_side
bar_fill_color': '889654', 'profile_link_color': '0021B3', 'default_profile': Fa
lse, 'statuses_count': 3344, 'profile_background_color': 'FFFFFF', 'profile_imag
e_url': 'http://pbs.twimg.com/profile_images/496692905335984128/XJh_d5f5_normal.
jpeg', 'profile_background_tile': True, 'id': 14983299, 'friends_count': 409, 'p
rofile_image_url_https': 'https://pbs.twimg.com/profile_images/49669290533598412
8/XJh_d5f5_normal.jpeg', 'following': False, 'created_at': 'Mon Jun 02 18:35:1
8 +0000 2008', 'is_translator': False, 'geo_enabled': True, 'is_translation_enabl
ed': False, 'follow_request_sent': False, 'followers_count': 2085, 'utc_offset'
: -18000, 'verified': False, 'profile_text_color': '383838', 'notifications': F
alse, 'entities': {'description': {'urls': []}, 'url': {'urls': [{'indices': [
0, 22], 'url': 'http://t.co/FM6dHXloIw', 'expanded_url': 'http://ryanemitchell.
com', 'display_url': 'ryanemitchell.com'}]}}, 'listed_count': 22, 'profile_banne
r_url': 'https://pbs.twimg.com/profile_banners/14983299/1412961553'}, 'retweeted
': False, 'in_reply_to_status_id_str': None, 'source': '<a href="http://ryanemit
chell.com" rel="nofollow">PythonScraping</a>', 'favorite_count': 0, 'text': 'Hell
o,world!', 'truncated': False, 'id': 538956506478428160, 'retweet_count': 0, 'fa
vorited': False, 'in_reply_to_status_id': None, 'geo': None, 'entities': {'user_m
entions': [], 'hashtags': [], 'urls': [], 'symbols': []}, 'coordinates': None, '
contributors': None}
```


这就是发了一篇推文的结果。我有时觉得 Twitter 之所以要限制 API 访问次数，是因为每个推文的字节很多，请求响应实在太费流量。

对于获取一组推文的请求，你可以通过设置推文数量来限制条数：

```
pythonStatuses = t.statuses.user_timeline(screen_name="montypython", count=5)
print(pythonStatuses)
```

这个例子中，我们请求 @montypython 推文中（也包括转发的推文）按时间排序最靠前的 5 条推文。

尽管这三个例子介绍了 Twitter API 的许多功能（搜索推文，获取任意用户的推文，用自己的账号发推文），但是 Twitter Python 库的能力远不止这些。你还可以搜索和操作 Twitter 的信息列表，已关注和未关注的用户，以及查看用户的简介信息，等等。完整的文档请在 GitHub（https://github.com/sixohsix/twitter）上查看。

4.6 Google API

Google 是目前为网民提供最全面、最好用的网络 API 套件（collection）的公司之一。无论你想处理哪种信息，包括语言翻译、地理位置、日历，甚至基因数据，Google 都提供了 API。Google 还为它的一些知名应用提供 API，比如 Gmail、YouTube 和 Blogger 等。

查看 Google API 有两种方式。一种方式是通过产品页面（https://developers.google.com/products/）查看，里面有许多 API、软件开发工具包，以及其他软件开发者感兴趣的项目。另一种方式是 API 控制台（https://console.developers.google.com/），里面提供了方便的接口来开启和关闭 API 服务，查看流量限制和使用情况，还可以和 Google 强大的云计算平台的开发实例结合使用。

Google 的大多数 API 都是免费的，不过有些需要付费，比如搜索 API 需要一个付费的授权。Google 的免费 API 套件对普通版的账号也是非常慷慨的，允许每天进行 250 次到 20 000 000 次的访问。还有一些 API 可以通过验证信用卡提高流量上限（不需要支付费用）。比如，Google 的地点查询 API 每 24 小时的流量限制是 1000 次，但是如果你通过了信用卡验证，就可以提高到 150 000 次。更多的信息请参考 Google 的 API 使用限额和计费方式页面（https://developers.google.com/places/webservice/usage）。

4.6.1 开始

如果你有 Google 账号，可以查看自己可用的 API 列表，并通过 Google 开发者控制台（https://console.developers.google.com/）创建 API 的 key。如果你没有 Google 账号，请在创建 Google 账号页面（https://accounts.google.com/SignUp）建立自己的账号。

当你登录账号或账号创建完成后，就能在 API 控制台页面（https://console.developers.google.com/project/201151233021/apiui/）看到一些账号信息，包含 API 的 key。单击左边菜单的“Credentials”（凭证）选项（图 4-3）：

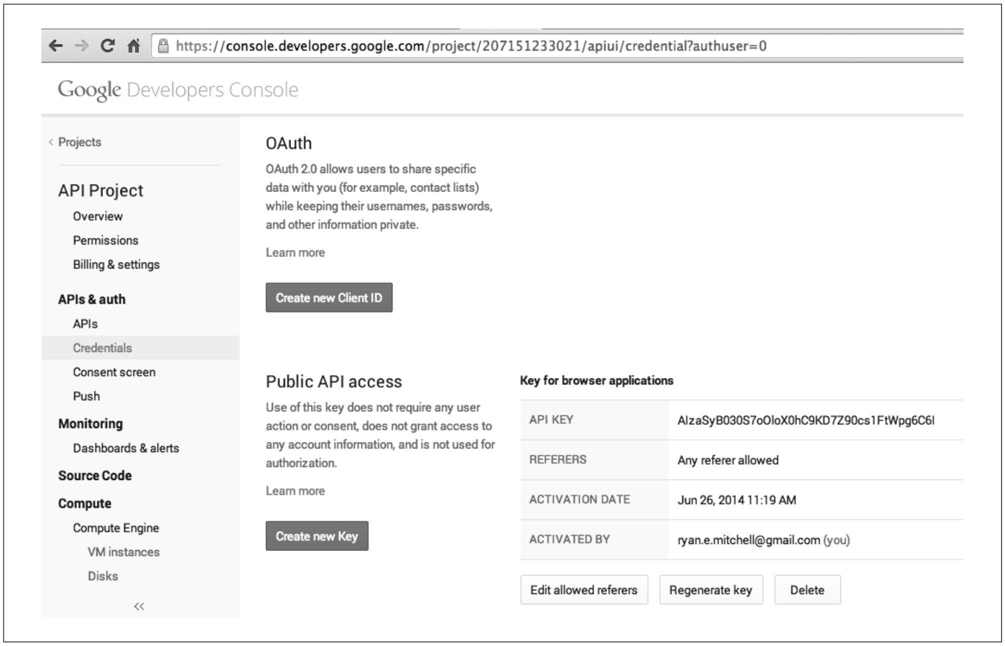


图 4-3：Google 的 API 凭证页面

在凭证页面，你可以单击“Create new Key”按钮创建新的 API key。为了你的账号安全，建议限制 API 使用的 IP 地址或 URL 链接。你也可以创建一个可用于任意 IP 地址或 URL 的 API key，只要把“Accept Request From These Server IP Addresses”（接受这些服务器 IP 地址发出的请求）这一栏空着就行了。但是，请记住保证 API key 的安全性是非常重要的事情，如果你不限制允许使用 API 的 IP 地址——任何使用你的 API key 调用你的 API 都算成是你的消费，即使你并不知情。

你也可以建立多个 API key。比如，你可以为每个项目都分配一个单独的 API key，也可以为每个网站域名都分配一个 API key。但是，Google 的 API 流量是按照每个账号分配的，不是按照每个 key 分配的，所以这样做虽然可以方便地管理 API 权限，但是并不会提高你的可用流量！

4.6.2　几个示例

Google 最受欢迎的（个人认为也是最有趣的）API 都在 Google 地图 API 套件中。你可能见过很多网站都在用嵌入式 Google 地图，觉得自己对这类功能很熟悉。但是，地图 API 远比嵌入式地图的功能丰富得多——你可以把街道地址解析成经 / 纬度（longitude/latitude）坐标值，地球上任意点的海拔高度，做出基于位置的可视化图形，获取任意位置的时区信息，以及其他一些地图相关的事情。

在你自己尝试这些例子的时候，请从 Google 的 API 控制台里把对应的 API 激活。Google 会把这些 API 的激活量作为应用量度（metric，即“统计多少用户在使用这个 API”），所以在使用这些 API 之前你需要激活它们。

用 Google 的 Geocode（地理位置信息）API 你可以在浏览器里实现一个简单的 GET 请求，把街道地址（这里用的是 Boston Museum of Science，里面有 Science Park）解析成纬度和经度：

```
https://maps.googleapis.com/maps/api/geocode/json?address=1+Science+Park+Boston+MA+
02114&key=<你的API key>
```

服务器响应的结果是：


```
"results" : [ { "address_components" : [ { "long_name" : "Museum Of Science Drive
way", "short_name" : "Museum Of Science Driveway", "types" : [ "route" ] }, { "l
ong_name" : "Boston", "short_name" : "Boston", "types" : [ "locality", "politica
l" ] }, { "long_name" : "Massachusetts", "short_name" : "MA", "types" : [ "admin
istrative_area_level_1", "political" ] }, { "long_name" : "United States", "shor
t_name" : "US", "types" : [ "country", "political" ] }, { "long_name" : "0211
4", "short_name" : "02114", "types" : [ "postal_code" ] } ], "formatted_address"
: "Museum Of Science Driveway, Boston, MA 02114, USA", "geometry" : { "bounds" :
{ "northeast" : { "lat" : 42.368454, "lng" : -71.06961339999999 }, "southwest" :
{ "lat" : 42.3672568, "lng" : -71.0719624 } }, "location" : { "lat" : 42.3677994
, "lng" : -71.0708078 }, "location_type" : "GEOMETRIC_CENTER", "viewport" : { "n
ortheast" : { "lat" : 42.3692043802915, "lng" : -71.06943891970849 }, "southwest
" : { "lat" : 42.3665064197085, "lng" : -71.0721368802915 } } }, "types" : [ "ro
ute" ] } ], "status" : "OK" }
```


其实我们在 API 里发送的地址格式并不好。但是 Google 就是 Google，Geocode API 会自动处理这些没有邮政编码或者州信息（甚至写错了）的地址，然后给你反馈标准的地址信息。例如，用错误的参数 `1+Skience+Park+Bostton+MA`（甚至没邮编）也会反馈同样的结果。

我曾经在一些任务中使用过 Geocode API，不仅是对用户在网站上填的地址进行标准化，还有采集网站上看着像地址的信息，用 API 对它们重新处理，形成更容易存储和搜索的数据。

你还可以用 Time zone（时区）API 获取任意经纬度的时区信息：

```
https://maps.googleapis.com/maps/api/timezone/json?location=42.3677994,-71.0708
078&timestamp=1412649030&key=<你的 API key>
```

服务器响应的结果是：


```
{ "dstOffset" : 3600, "rawOffset" : -18000, "status" : "OK", "timeZon
eId" : "America/New_York", "timeZoneName" : "Eastern Daylight Time" }
```


Time zone API 需要用一个 Unix 时间戳才能发出请求。它可以让 Google 为你提供一个经过

时区调整的夏令时。即使在那些时区不受夏令时影响的地区（比如凤凰城不实行夏令时[7]），API 请求也是需要时间戳的。

你可以用地点经纬度获取对应的海拔高度，来结束我们短暂的 Google 地图 API 旅程：

```
https://maps.googleapis.com/maps/api/elevation/json?locations=42.3677994,-71.070
8078&key=<你的 API key>
```

返回的结果是该地点海平面以上的海拔高度（单位是米），其中一个参数“resolution”（分辨率）的数值表示对这个点进行插值计算的海拔高度的样本数据中与这个点最远的距离（单位是米）。这个值越小，表示计算出的海拔高度的值越精确。


```
{ "results" : [ { "elevation" : 5.127755641937256, "location" : { "la
t" : 42.3677994, "lng" : -71.0708078 }, "resolution" : 9.543951988220
215 } ], "status" : "OK" }
```


4.7　解析JSON数据

在本章中，我们介绍了许多不同类型的 API 以及它们的使用方法，也介绍了这些 API 反馈的一些简单的 JSON 格式数据。现在让我们看看如何解析和使用这些信息。

本章开始的时候，我用过 freegeoip.net 网站 IP 查询的例子，可以把 IP 地址解析转换成地理位置：

```
http://freegeoip.net/json/50.78.253.58
```

我可以获取这个请求的反馈数据，然后用 Python 的 JSON 解析函数来解码：

```
import json
from urllib.request import urlopen

def getCountry(ipAddress):
    response = urlopen("http://freegeoip.net/json/"+ipAddress).read()
                     .decode('utf-8')
    responseJson = json.loads(response)
    return responseJson.get("country_code")

print(getCountry("50.78.253.58"))
```

这段代码可以打印出 IP 地址为 50.78.253.58 的国家代码。

这里用的 JSON 解析库是 Python 标准库的一部分。只需要在代码开头写上 `import json`，你就可以使用它了！不同于那些需要先把 JSON 解析成一种 JSON 对象或 JSON 节点的语言，Python 使用了一种更加灵活的方式，把 JSON 转换成字典，JSON 数组转换成列表，

注 7：美国亚利桑那州全州不用夏令时，凤凰城是其州府。——译者注

JSON 字符串转换成 Python 字符串。通过这种方式，就可以让 JSON 的获取和操作变得非常简单。

下面的例子演示了如何使用 Python 的 JSON 解析库，处理 JSON 字符串中可能出现的不同数据类型：

```
import json

jsonString = '{"arrayOfNums":[{"number":0},{"number":1},{"number":2}],
               "arrayOfFruits":[{"fruit":"apple"},{"fruit":"banana"},
                                {"fruit":"pear"}]}'
jsonObj = json.loads(jsonString)

print(jsonObj.get("arrayOfNums"))
print(jsonObj.get("arrayOfNums")[1])
print(jsonObj.get("arrayOfNums")[1].get("number")+
      jsonObj.get("arrayOfNums")[2].get("number"))
print(jsonObj.get("arrayOfFruits")[2].get("fruit"))
```

输出的结果是：

```
[{'number': 0}, {'number': 1}, {'number': 2}]
{'number': 1}
3
pear
```

第一行是一个组词典构成的列表对象，第二行是一个词典对象，第三行是一个整数（第一行词典列表里整数的和），第四行是一个字符串。

4.8 回到主题

虽然，有一些新式的网络应用存在的理由（raison d'être）就是采集现有的数据，再用更好看的形式展现出来，但是我觉得这些应用没什么意义。如果你用 API 作为唯一的数据源，那么你最多就是复制别人数据库里的数据，不过都是些已经公布过的“黄花菜”。真正有意思的事情，是把多个数据源组合成新的形式，或者把 API 作为一种工具，从全新的视角对采集到的数据进行解释。

下面介绍如何把 API 和网络数据采集结合起来：看看维基百科的贡献者们大都在哪里。

如果你经常用维基百科，可能会注意到词条的编辑历史页面，里面是一列编辑记录。如果用户先登入维基百科再编辑词条，他们的用户名就会显示出来。如果不先登录就对词条进行编辑，他们的 IP 地址就会显示在编辑历史中，如图 4-4 所示。

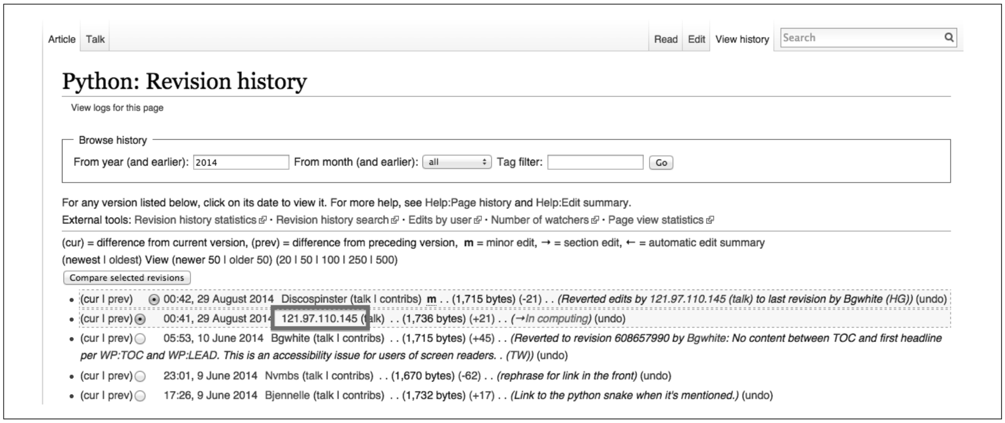


图 4-4：维基百科 Python 词条的编辑历史页面的匿名编辑者的 IP 地址

上图中我们标注的 IP 地址是 121.97.110.145。用 freegeoip.net 的 API，我们可以查出这个 IP 地址的地理位置（IP 地址有时会改变地理位置）是在菲律宾（Phillipines）的奎松市（Quezon）。

一个这样的 IP 地址并没什么意义，但是如果我们可以收集大量维基百科编辑者的地理数据呢？几年前我做过这件事儿，当时用 Google 的地理图形库（Geochart library，https://developers.google.com/chart/interactive/docs/gallery/geochart）做了一个显示维基百科英文版的编辑者所在位置的可视图，后来又做了其他语言的版本，如图 4-5 所示。



图 4-5：用 Google 的 Geochart 库创建的维基百科编辑者地理位置可视化

首先做一个采集维基百科的基本程序，寻找编辑历史页面，然后把编辑历史里面的 IP 地址找出来，这并不难。只要对第 3 章的代码做些修改就可以，代码如下所示：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import datetime
import random
import re

random.seed(datetime.datetime.now())
def getLinks(articleUrl):
    html = urlopen("http://en.wikipedia.org"+articleUrl)
    bsObj = BeautifulSoup(html)
    return bsObj.find("div", {"id":"bodyContent"}).findAll("a",
                     href=re.compile("^(/wiki/)((?!:).)*$"))

def getHistoryIPs(pageUrl):
    # 编辑历史页面URL链接格式是：
    # http://en.wikipedia.org/w/index.php?title=Title_in_URL&action=history
    pageUrl = pageUrl.replace("/wiki/", "")
    historyUrl = "http://en.wikipedia.org/w/index.php?title="
                 +pageUrl+"&action=history"
    print("history url is: "+historyUrl)
    html = urlopen(historyUrl)
    bsObj = BeautifulSoup(html)
    # 找出class属性是"mw-anonuserlink"的链接
    # 它们用IP地址代替用户名
    ipAddresses = bsObj.findAll("a", {"class":"mw-anonuserlink"})
    addressList = set()
    for ipAddress in ipAddresses:
        addressList.add(ipAddress.get_text())
    return addressList

links = getLinks("/wiki/Python_(programming_language)")

while(len(links) > 0):
    for link in links:
        print("-------------------")
        historyIPs = getHistoryIPs(link.attrs["href"])
        for historyIP in historyIPs:
            print(historyIP)

    newLink = links[random.randint(0, len(links)-1)].attrs["href"]
    links = getLinks(newLink)
```

这个程序包括两个函数：`getLinks`（第 3 章里用过）和新的函数 `getHistoryIPs`，搜索所有 `mw-anonuserlin` 类里面的链接信息（匿名用户的 IP 地址，不是用户名），返回一个链接列表。

Python 的集合类型简介

到现在为止，我用已经用过两个 Python 的数据结构来储存不同类型的数据：列表和词典。已经有了两种数据类型，为什么还要用集合（set）？

Python 的集合是无序的，就是说你不能用位置来获得集合元素对应的值。数据加入集合的顺序，和你重新获取它们的顺序，很可能是不一样的。在上面的示例代码中，使用集合的一个好处就是它不会储存重复值。如果你要存储一个已有的值到集合中，集合会自动忽略它。因此，我们可以快速地获取历史编辑页面中独立的 IP 地址，不需要考虑同一个编辑者多次编辑历史的情况。

对于未来可能需要扩展的代码，在决定使用集合还是列表时，有两件事情需要考虑：虽然列表迭代速度比集合稍微快一点儿，但集合查找速度更快（确定一个对象是否在集合中），因为 Python 集合就是值为 `None` 的词典，用的是哈希表结构，查询速度为 $O(1)$。

上面的代码还用了一些随机的（不过对这个示例是有效的）搜索模式来查找词条的编辑历史。首先获取起始词条连接的所有词条的编辑历史（示例中是 Python programming language 词条）。然后，随机选择一个词条作为起始点，再获取这个页面连接的所有词条的编辑历史。重复这个过程直到页面没有连接维基词条为止。

现在，我们获得了编辑历史的 IP 地址数据，把它们与上一节的 `getCountry` 函数结合起来，就可以查询 IP 地址所属的国家和地区了。我对 `getCountry` 函数做了一点儿修改，处理了无效或错误的 IP 地址引起的“404 Not Found”异常（比如，写到这里时，freegeoip.net 不能查询 IPv6 地址，可能会引起 404 错误）：

```
def getCountry(ipAddress):
    try:
        response = urlopen("http://freegeoip.net/json/"
                               +ipAddress).read().decode('utf-8')
    except HTTPError:
        return None
    responseJson = json.loads(response)
    return responseJson.get("country_code")

links = getLinks("/wiki/Python_(programming_language)")


while(len(links) > 0):
    for link in links:
        print("-------------------")
        historyIPs = getHistoryIPs(link.attrs["href"])
        for historyIP in historyIPs:
            country = getCountry(historyIP)
            if country is not None:
                print(historyIP+" is from "+country)
```

```
        newLink = links[random.randint(0, len(links)-1)].attrs["href"]
        links = getLinks(newLink)
```

完整代码在 http://www.pythonscraping.com/code/6-3.txt。下面是部分输出结果：

```
-------------------
history url is: http://en.wikipedia.org/w/index.php?title=Programming_
paradigm&action=history
68.183.108.13 is from US
86.155.0.186 is from GB
188.55.200.254 is from SA
108.221.18.208 is from US
141.117.232.168 is from CA
76.105.209.39 is from US
182.184.123.106 is from PK
212.219.47.52 is from GB
72.27.184.57 is from JM
49.147.183.43 is from PH
209.197.41.132 is from US
174.66.150.151 is from US
```

4.9　再说一点API

本章我们介绍了几个新式 API 常用的获取网络数据的方式，重点介绍了有助于网络数据采集工作的 API 用法。但是，对 API 的这点儿介绍还是远远不够的，API 的内容非常丰富，这里并没有体现出 API 具有“许多不同的软件都可以通过相同的 API 分享数据”的特点。

由于本书的主题是网络数据采集，因此无意成为数据收集的百科全书，如果你需要，我只能为你推荐一些优质的资源，帮助你对这个主题进行深入的研究。

Leonard Richardson、Mike Amundsen 和 Sam Ruby 的 *RESTful Web APIs*（http://shop.oreilly.com/product/0636920028468.do）为网络 API 的用法提供了非常全面的理论与实践指导。另外，Mike Amundsen 的精彩视频教学课程 Designing APIs for the Web（http://shop.oreilly.com/product/110000125.do），也可以教你创建自己的 API。如果你想把自己采集的数据用一种便捷的方式分享出来，他的视频非常有用。

虽然初看网络数据采集和网络 API 好像完全是两个不同的主题，但是希望这一章的内容可以为你呈现出两者在网络数据收集这个领域中相互补充的能力。从某种意义上看，网络 API 的使用可以作为网络数据采集的一个子集。毕竟，最终都是要从网络服务器收集数据，然后把它们解析成可用的数据格式，这和你用任何网络爬虫做的事情一模一样。

第 5 章

存储数据

虽然在命令行里显示运行结果很有意思，但是随着数据不断增多，并且需要进行数据分析时，将数据打印到命令行就不是办法了。为了可以远程使用大部分网络爬虫，你还需要把采集到的数据存储起来。

本章将介绍三种主要的数据管理方法，对绝大多数应用都适用。如果你准备创建一个网站的后端服务或者创建自己的 API，那么可能都需要让爬虫把数据写入数据库。如果你需要一个快速简单的方法收集网上的文档，然后存到你的硬盘里，那么可能需要创建一个文件流（file stream）来实现。如果还要为偶然事件提个醒儿，或者每天定时收集当天累计的数据，就给自己发一封邮件吧！

抛开与网络数据采集的关系，大数据存储和与数据交互的能力，在新式的程序开发中也已经是重中之重了。这一章的内容其实是实现第二部分许多示例的基础。如果你对自动数据存储相关的知识不太了解，我非常希望你至少能浏览一下。

5.1 媒体文件

存储媒体文件有两种主要的方式：只获取文件 URL 链接，或者直接把源文件下载下来。你可以通过媒体文件所在的 URL 链接直接引用它。这样做的优点如下。

- 爬虫运行得更快，耗费的流量更少，因为只要链接，不需要下载文件。
- 可以节省很多存储空间，因为只需要存储 URL 链接就可以。
- 存储 URL 的代码更容易写，也不需要实现文件下载代码。
- 不下载文件能够降低目标主机服务器的负载。

不过这么做也有一些缺点。

- 这些内嵌在你的网站或应用中的外站 URL 链接被称为盗链（hotlinking），使用盗链可能会让你麻烦不断，每个网站都会实施防盗链措施。
- 因为你的链接文件在别人的服务器上，所以你的应用就要跟着别人的节奏运行了。
- 盗链是很容易改变的。如果你把盗链图片放在博客上，要是被对方服务器发现，很可能被恶搞。如果你把 URL 链接存起来准备以后再用，可能用的时候链接已经失效了，或者是变成了完全无关的内容。
- 现实中的网络浏览器不仅可以请求 HTML 页面并切换页面，它们也会下载访问页面上所有的资源。下载文件会让你的爬虫看起来更像是人在浏览网站，这样做反而有好处。

如果你还在犹豫究竟是存储文件，还是只存储文件的 URL 链接，可以想想这些文件是要多次使用，还是放进数据库之后就只是等着“落灰”，再也不会被打开。如果答案是后者，那么最好还是只存储这些文件的 URL 吧。如果答案是前者，那么就继续往下看！

在 Python 3.x 版本中，`urllib.request.urlretrieve` 可以根据文件的 URL 下载文件：

```
from urllib.request import urlretrieve
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen("http://www.pythonscraping.com")
bsObj = BeautifulSoup(html)
imageLocation = bsObj.find("a", {"id": "logo"}).find("img")["src"]
urlretrieve (imageLocation, "logo.jpg")
```

这段程序从 http://pythonscraping.com 下载 logo 图片，然后在程序运行的文件夹里保存为 logo.jpg 文件。

如果你只需要下载一个文件，而且知道如何获取它，以及它的文件类型，这么做就可以了。但是大多数爬虫都不可能一天只下载一个文件。下面的程序会把 http://pythonscraping.com 主页上所有 `src` 属性的文件都下载下来：

```
import os
from urllib.request import urlretrieve
from urllib.request import urlopen
from bs4 import BeautifulSoup

downloadDirectory = "downloaded"
baseUrl = "http://pythonscraping.com"

def getAbsoluteURL(baseUrl, source):
    if source.startswith("http://www."):
        url = "http://"+source[11:]
    elif source.startswith("http://"):
        url = source
    elif source.startswith("www."):
```

```
        url = source[4:]
        url = "http://"+source
    else:
        url = baseUrl+"/"+source
    if baseUrl not in url:
        return None
    return url

def getDownloadPath(baseUrl, absoluteUrl, downloadDirectory):
    path = absoluteUrl.replace("www.", "")
    path = path.replace(baseUrl, "")
    path = downloadDirectory+path
    directory = os.path.dirname(path)

    if not os.path.exists(directory):
        os.makedirs(directory)

    return path

html = urlopen("http://www.pythonscraping.com")
bsObj = BeautifulSoup(html)
downloadList = bsObj.findAll(src=True)

for download in downloadList:
    fileUrl = getAbsoluteURL(baseUrl, download["src"])
    if fileUrl is not None:
        print(fileUrl)

urlretrieve(fileUrl, getDownloadPath(baseUrl, fileUrl, downloadDirectory))
```

程序运行注意事项

你知道从网上下载未知文件的那些警告吗？这个程序会把页面上所有的文件下载到你的硬盘里，可能会包含一些 bash 脚本、.exe 文件，甚至可能是恶意软件（malware）。

如果你之前从没有运行过任何下载到电脑里的文件，电脑就是安全的吗？尤其是当你用管理员权限运行这个程序时，你的电脑基本已经处于危险之中。如果你执行了网页上的一个文件，那个文件把自己传送到了 ../../../../usr/bin/python 里面，会发生什么呢？等下一次你再运行 Python 程序时，你的电脑就可能会安装恶意软件。

这个程序只是为了演示；请不要随意运行它，因为这里没有对所有下载文件的类型进行检查，也不应该用管理员权限运行它。记得经常备份重要的文件，不要在硬盘上存储敏感信息，小心驶得万年船。

这个程序首先使用 Lambda 函数（第 2 章介绍过）选择首页上所有带 `src` 属性的标签。然后对 URL 链接进行清理和标准化，获得文件的绝对路径（而且去掉了外链）。最后，每个

文件都会下载到程序所在文件夹的 downloaded 文件里。

这里 Python 的 os 模块用来获取每个下载文件的目标文件夹，建立完整的路径。os 模块是 Python 与操作系统进行交互的接口，它可以操作文件路径，创建目录，获取运行进程和环境变量的信息，以及其他系统相关的操作。

5.2 把数据存储到CSV

CSV（Comma-Separated Values，逗号分隔值）是存储表格数据的常用文件格式。Microsoft Excel 和很多应用都支持 CSV 格式，因为它很简洁。下面就是一个 CSV 文件的例子：

```
fruit,cost
apple,1.00
banana,0.30
pear,1.25
```

和 Python 一样，CSV 里留白（whitespace）也是很重要的：每一行都用一个换行符分隔，列与列之间用逗号分隔（因此也叫“逗号分隔值”）。CSV 文件还可以用 Tab 字符或其他字符分隔行，但是不太常见，用得不多。

如果你只想从网页上把 CSV 文件下载到电脑里，不打算做任何解析和修改，那么这节后面的内容就没必要再看了。只要用上一节里介绍的文件下载方法下载并保存为 CSV 格式就行了。

Python 的 csv 库可以非常简单地修改 CSV 文件，甚至从零开始创建一个 CSV 文件：

```
import csv

csvFile = open("../files/test.csv", 'w+')
try:
    writer = csv.writer(csvFile)
    writer.writerow(('number', 'number plus 2', 'number times 2'))
    for i in range(10):
        writer.writerow( (i, i+2, i*2))
finally:
    csvFile.close()
```

这里提个醒儿：Python 新建文件的机制考虑得非常周到（bullet-proof）。如果 ../files/test.csv 不存在，Python 会自动创建文件（不会自动创建文件夹）。如果文件已经存在，Python 会用新的数据覆盖 test.csv 文件。

运行完成后，你会看到一个 CSV 文件：

```
number,number plus 2,number times 2
0,2,0
1,3,2
```

```
2,4,4
...
```

网络数据采集的一个常用功能就是获取 HTML 表格并写入 CSV 文件。维基百科的文本编辑器对比词条（https://en.wikipedia.org/wiki/Comparison_of_text_editors）中用了许多复杂的 HTML 表格，用到了颜色、链接、排序，以及其他在写入 CSV 文件之前需要忽略的 HTML 元素。用 BeautifulSoup 和 `get_text()` 函数，你可以用十几行代码完成这件事：

```
import csv
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen("http://en.wikipedia.org/wiki/Comparison_of_text_editors")
bsObj = BeautifulSoup(html)
# 主对比表格是当前页面上的第一个表格
table = bsObj.findAll("table",{"class":"wikitable"})[0]
rows = table.findAll("tr")

csvFile = open("../files/editors.csv", 'wt', newline='', encoding='utf-8')
writer = csv.writer(csvFile)
try:
    for row in rows:
    csvRow = []
    for cell in row.findAll(['td', 'th']):
        csvRow.append(cell.get_text())
        writer.writerow(csvRow)
finally:
    csvFile.close()
```

实际工作中写此程序之前的注意事项

如果你有很多 HTML 表格，且每个都要转换成 CSV 文件，或者许多 HTML 表格都要汇总到一个 CSV 文件，那么把这个程序整合到爬虫里以解决问题非常好。但是，如果你只需要做一次这种事情，那么更好的办法就是：复制粘贴。选择 HTML 表格内容然后粘贴到 Excel 文件里，可以另存为 CSV 格式，不需要写代码就能搞定！

这个程序会在程序上一层目录的 files 文件夹里生成一个 CSV 文件 ../files/editors.csv——把这个程序分享给那些不熟悉 MySQL 的朋友们吧！

5.3 MySQL

MySQL（官方发音是“My es-kew-el”，但很多人都说成“My Sequel”）是目前最受欢迎的开源关系型数据库管理系统。一个开源项目具有如此之竞争力实在是令人意外，它的流行程度正在不断地接近另外两个闭源的商业数据库系统：微软的 SQL Server 和甲骨文的 Oracle 数据库（MySQL 在 2010 年被甲骨文收购）。

它的流程程度实在是名符其实。对大多数应用来说，MySQL 都是不二选择。它是一种非常灵活、稳定、功能齐全的 DBMS，许多顶级的网站都在用它：YouTube[1]、Twitter[2] 和 Facebook[3] 等。

因为它受众广泛，免费，开箱即用，所以它也是网络数据采集项目中常用的数据库，我们将在本书后面的示例中使用它。

> **“关系型”数据库？**
>
> “关系型数据”就是有关联的数据。就是这么简单！
>
> 开个玩笑！当计算机科学家说起关系型数据时，他们指的是那些并非孤立的数据——它们的属性与其他的数据是有关联的。例如，“用户 A 在学校 B 上学”，这里用户 A 在数据库的“用户”表中，而学校 B 是在数据库的“学校”表中。
>
> 在本章后面的内容里，我们将介绍数据关系的不同类型，以及如何有效地把数据存储到 MySQL（或其他关系型数据库）里。

5.3.1 安装MySQL

如果你第一次接触 MySQL，安装数据库听着可能有点儿吓人（如果你是老手，可以跳过这部分内容）。其实，安装方法和安装其他软件一样简单。归根到底，MySQL 就是由一系列数据文件构成的，储存在你的远端服务器或本地的电脑上，里面包含了数据库存储的所有信息。MySQL 软件层提供了一种与数据交互的便捷操作方法。例如，下面的命令把用户表 users 中名字为“Ryan”的用户找出来：

```
SELECT * FROM users WHERE firstname = "Ryan"
```

如果你用 Ubuntu（或其他 Debain 分支系统），安装 MySQL 很简单：

```
$sudo apt-get install mysql-server
```

只要稍微留意一下安装过程，看看电脑是不是可以满足安装的内存需求，然后在安装提示的地方为 root 用户设置新密码就可以了。

Mac OS X 和 Windows 系统的安装过程有点儿复杂。如果你没有甲骨文账户，下载 MySQL 安装包之前需要先注册一下。

注 1：Joab Jackson, “YouTube Scales MySQL with Go Code,” PCWorld, December 15, 2012 (http://bit.ly/1LWVmc8)。

注 2：Jeremy Cole and Davi Arnaut, “MySQL at Twitter,” The Twitter Engineering Blog, April 9, 2012 (http://bit.ly/1KHDKns)。

注 3：“MySQL and Database Engineering: Mark Callaghan,” March 4, 2012（http://on.fb.me/1RFMqvw）。

如果你用 Mac OS X 系统，请先下载对应的安装包（http://dev.mysql.com/downloads/mysql/）。

选择 .dmg 安装包，登入网站或者创建一个账户，开始下载文件。下载完成后打开安装包，你会看到一个简单的安装向导（图 5-1）。

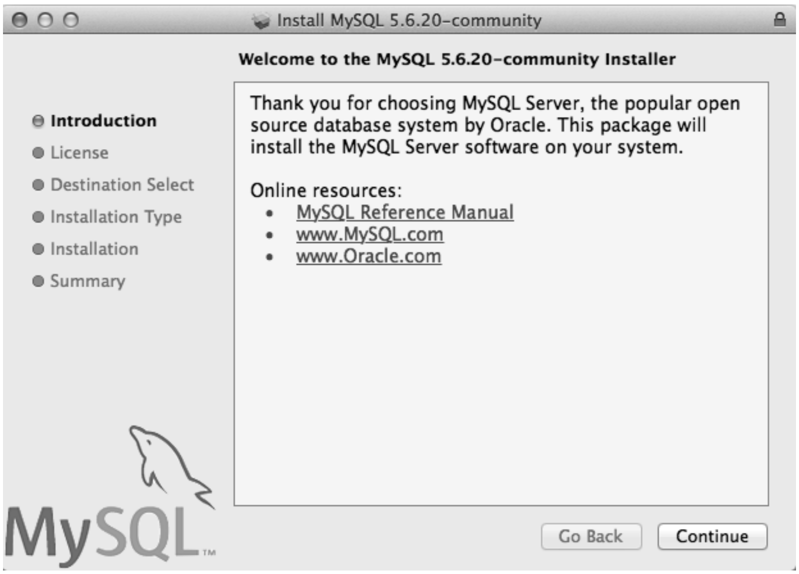


图 5-1：Mac OS X 的 MySQL 安装工具

使用默认安装步骤就可以，本书后面使用 MySQL 都是假设你用的默认安装步骤。

如果觉得下载安装包再执行安装工具太无聊，也可以用 Mac OS X 的包管理器 Homebrew（http://brew.sh/）安装。当 Homebrew 安装好以后，用下面的命令安装 MySQL：

```
$brew install mysql
```

Homebrew 是一个伟大的开源工具，与 Python 包完美结合。其实，本书使用的大多数 Python 第三方库都可以用 Homebrew 安装。如果你还没用过，强烈推荐你试一下！

Mac OS X 的 MySQL 安装好之后，你可以用下面的命令启动 MySQL 服务器：

```
$cd /usr/local/mysql
$sudo ./bin/mysqld_safe
```

在 Windows 系统上，安装和运行 MySQL 更复杂一些，但是有个方便的安装工具（http://dev.mysql.com/downloads/windows/installer/）可以简化这个过程（图 5-2）。



图 5-2：MySQL Windows 安装工具 .png

用默认选项安装 MySQL 就可以，不过有一个地方要注意：在 Setup Type（类型设置）页面，建议你选择“Server Only”（只选服务器）选项，这可以避免安装一堆微软的软件和库文件。

然后，你就可以用默认设置安装，跟着提示一步步操作就可以启动 MySQL 服务器了。

5.3.2　基本命令

MySQL 服务器启动之后，有很多种方法可以与数据库交互。因为有很多工具是图形界面，所以你不用 MySQL 的命令行（或者很少用命令行）也能管理数据库。像 phpMyAdmin 和 MySQL Workbench 这类工具都可以很容易地实现数据的查看、排序和新建等工作。但是，掌握命令行操作数据库依然是很重要的。

除了用户自定义变量名（MySQL 5.x 版本是不区分大小写的，MySQL 5.0 之前的版本是不区分大小写的），MySQL 语句是不区分大小写的。例如，SELECT 和 sElEcT 是一样的，不过习惯上写 MySQL 语句的时候所有的 MySQL 关键词都用大写。大多数开发者还喜欢用小写字母表示数据表和数据库的名称，虽然这个标准经常不被注意。

当你首次登入 MySQL 的时候，里面是没有数据库存放数据的。你可以创建一个：

```
>CREATE DATABASE scraping;
```

因为每个 MySQL 实例可以有多个数据库，所以使用某个数据库之前需要指定数据库的名称：

```
>USE scraping;
```

从现在开始（直到关闭 MySQL 链接或切换到另一个数据库之前），所有的命令都运行在这个新的“scraping”数据库里面。

所有操作看着都非常简单。那么，在数据库里创建数据表的方法应该也类似吧？让我们在数据库里创建一个表来存储采集的网页：

```
>CREATE TABLE pages;
```

结果显示错误：

```
ERROR 1113 (42000): A table must have at least 1 column
```

和数据库不同，MySQL 数据表必须至少有一列，否则不能创建。为了在 MySQL 里定义字段（数据列），你必须在 CREATE TABLE <tablename> 语句后面，把字段的定义放进一个带括号的、内部由逗号分隔的列表中：

```
>CREATE TABLE pages (id BIGINT(7) NOT NULL AUTO_INCREMENT, title VARCHAR(200),
content VARCHAR(10000), created TIMESTAMP DEFAULT CURRENT_TIMESTAMP, PRIMARY KEY
(id));
```

每个字段定义由三部分组成：

- 名称（id、title、created 等）
- 数据类型（BIGINT(7)、VARCHAR、TIMESTAMP）
- 其他可选属性（NOT NULL AUTO_INCREMENT）

在字段定义列表的最后，还要定义一个“主键”（key）。MySQL 用这个主键来组织表的内容，便于后面快速查询。在本章后面的内容里，我将介绍如何调整这些主键以提高数据库的查询速度，但是现在，用表的 id 列作为主键就可以。

语句执行之后，你可以用 DESCRIBE 查看数据表的结构：

```
> DESCRIBE pages;
+---------+----------------+------+-----+-------------------+----------------+
| Field   | Type           | Null | Key | Default           | Extra          |
+---------+----------------+------+-----+-------------------+----------------+
| id      | bigint(7)      | NO   | PRI | NULL              | auto_increment |
| title   | varchar(200)   | YES  |     | NULL              |                |
```

```
| content | varchar(10000) | YES  |     | NULL              |                |
| created | timestamp      | NO   |     | CURRENT_TIMESTAMP |                |
+---------+----------------+------+-----+-------------------+----------------+
4 rows in set (0.00 sec)
```

当然，这还是一个空表。你可以在 pages 表里插入一些测试数据，如下所示：

```
> INSERT INTO pages (title, content) VALUES ("Test page title", "This is some te
st page content. It can be up to 10,000 characters long.");
```

需要注意的是，虽然 pages 表里有四个字段（id、title、content、created），但实际上你只需要插入两个字段（title 和 content）的数据即可。因为 id 字段是自动递增的（每次新插入数据行时 MySQL 自动增加 1），通常不用处理。另外，created 字段的类型是 timestamp，默认就是数据加入时的时间戳。

当然，我们也可以自定义四个字段的内容：

```
INSERT INTO pages (id, title, content, created) VALUES (3, "Test page title", "
This is some test page content. It can be up to 10,000 characters long.", "2014-
09-21 10:25:32");
```

只要你定义的整数在数据表的 id 字段里没有，它就可以顺利插入数据表。但是，这么做非常不好；除非万不得已（比如程序中断漏了一行数据），否则让 MySQL 自己处理 id 和 timestamp 字段。

现在表里有一些数据了，你可以用很多方法来选择这些数据。下面是几个 SELECT 语句的示例：

```
>SELECT * FROM pages WHERE id = 2;
```

这条语句告诉 MySQL，“从 pages 表中把 id 字段中等于 2 的整行数据全挑选出来”。这个星号（*）是通配符，表示所有字段，这行语句会把满足条件（WHERE id = 2）的所有字段都显示出来。如果 id 字段里没有任何一行等于 2，就会返回一个空集。例如，下面这个不区分大小写的查询，会返回 title 字段里包含“test”的所有行（% 符合表示 MySQL 字符串通配符）的所有字段：

```
>SELECT * FROM pages WHERE title LIKE "%test%";
```

但是，如果你的表有很多字段，而你只想返回部分字段怎么办？你可以不用星号，而用下面的方式：

```
>SELECT id, title FROM pages WHERE content LIKE "%page content%";
```

这样就只会返回 title 字段包含“page content”的所有行的 id 和 title 两个字段了。

DELETE 语句语法与 SELECT 语句类似：

```
>DELETE FROM pages WHERE id = 1;
```

由于数据库的数据删除后不能恢复，所以在执行 DELETE 语句之前，建议用 SELECT 确认一下要删除的数据（本例中，就是用 SELECT * FROM pages WHERE id = 1 查看），然后把 SELECT * 换成 DELETE 就可以了，这会是一个好习惯。很多程序员都有过一些 DELETE 误操作的伤心往事，还有一些恐怖故事就是有人慌乱中忘了在语句中放 WHERE，结果把所有客户数据都删除了。别让这种事发生在你身上！

还有一个需要介绍的语句是 UPDATE：

```
>UPDATE pages SET title="A new title", content="Some new content" WHERE id=2;
```

结合本书的主题，后面我们就只用这些基本的 MySQL 语句，做一些简单的数据查询、创建和更新工作。如果你对这个强大数据库的命令和技术感兴趣，推荐你去看 Paul DuBois 的 *MySQL Cookbook*（http://shop.oreilly.com/product/0636920032274.do）。

5.3.3　与Python整合

Python 没有内置的 MySQL 支持工具。不过，有很多开源的库可以用来与 MySQL 做交互，Python 2.x 和 Python 3.x 版本都支持。最有名的一个库就是 PyMySQL（https://github.com/PyMySQL/PyMySQL）。

写到这里的时候，PyMySQL 的版本是 0.6.2，你可以用下面的命令下载并安装它：

```
$ curl -L https://github.com/PyMySQL/PyMySQL/tarball/pymysql-0.6.2 | tar xz
$ cd PyMySQL-PyMySQL-f953785/
$ python setup.py install
```

如果需要更新，请检查最新版的 PyMySQL，并修改第一行下载链接中的版本号进行更新。

安装完成之后，你就可以使用 PyMySQL 包了。如果你的 MySQL 服务器处于运行状态，应该就可以成功地执行下面的命令（记得把 root 账户密码加进去）：

```
import pymysql
conn = pymysql.connect(host='127.0.0.1', unix_socket='/tmp/mysql.sock',
                       user='root', passwd=None, db='mysql')
cur = conn.cursor()
cur.execute("USE scraping")

cur.execute("SELECT * FROM pages WHERE id=1")
print(cur.fetchone())
cur.close()
conn.close()
```

这段程序有两个对象：连接对象（conn）和光标对象（cur）。

连接 / 光标模式是数据库编程中常用的模式，不过刚刚接触数据库的时候，有些用户很难区分两种模式的不同。连接模式除了要连接数据库之外，还要发送数据库信息，处理回滚操作（当一个查询或一组查询被中断时，数据库需要回到初始状态，一般用事务控制手段实现状态回滚），创建新的光标对象，等等。

而一个连接可以有很多个光标。一个光标跟踪一种状态（state）信息，比如跟踪数据库的使用状态。如果你有多个数据库，且需要向所有数据库写内容，就需要多个光标来处理。光标还会包含最后一次查询执行的结果。通过调用光标函数，比如 `cur.fetchone()`，可以获取查询结果。

用完光标和连接之后，千万记得把它们关闭。如果不关闭就会导致*连接泄漏*（connection leak），造成一种未关闭连接现象，即连接已经不再使用，但是数据库却不能关闭，因为数据库不能确定你还要不要继续使用它。这种现象会一直耗费数据库的资源，所以用完数据库之后记得关闭连接！

刚开始的时候，你最想做的事情可能就是把采集的结果保存到数据库里。让我们用前面维基百科爬虫的例子来演示一下如何实现数据存储。

在进行网络数据采集时，处理 Unicode 字符串是很痛苦的事情。默认情况下，MySQL 也不支持 Unicode 字符处理。不过你可以设置这个功能（这么做会增加数据库的占用空间）。因为在维基百科上我们难免会遇到各种各样的字符，所以最好一开始就让你的数据库支持 Unicode：

```
ALTER DATABASE scraping CHARACTER SET = utf8mb4 COLLATE = utf8mb4_unicode_ci;
ALTER TABLE pages CONVERT TO CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
ALTER TABLE pages CHANGE title title VARCHAR(200) CHARACTER SET utf8mb4 COLLATE
utf8mb4_unicode_ci;
ALTER TABLE pages CHANGE content content VARCHAR(10000) CHARACTER SET utf8mb4 CO
LLATE utf8mb4_unicode_ci;
```

这四行语句改变的内容有：数据库、数据表，以及两个字段的默认编码都从 `utf8mb4`（严格说来也属于 Unicode，但是对大多数 Unicode 字符的支持都非常不好）转变成了 `utf8mb4_unicode_ci`。

你可以在 `title` 或 `content` 字段中插入一些德语变音符（umlauts）或汉语字符，如果没有错误就表示转换成功了。

现在数据库已经准备好接收维基百科的各种信息了，你可以用下面的程序来存储数据：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
import datetime
import random
```

```python
import pymysql

conn = pymysql.connect(host='127.0.0.1', unix_socket='/tmp/mysql.sock',
                       user='root', passwd=None, db='mysql', charset='utf8')
cur = conn.cursor()
cur.execute("USE scraping")

random.seed(datetime.datetime.now())

def store(title, content):
    cur.execute("INSERT INTO pages (title, content) VALUES (\"%s\",
            \"%s\")", (title, content))
    cur.connection.commit()

def getLinks(articleUrl):
    html = urlopen("http://en.wikipedia.org"+articleUrl)
    bsObj = BeautifulSoup(html)
    title = bsObj.find("h1").get_text()
    content = bsObj.find("div", {"id":"mw-content-text"}).find("p").get_text()
    store(title, content)
    return bsObj.find("div", {"id":"bodyContent"}).findAll("a",
                      href=re.compile("^(/wiki/)((?!:).)*$"))

links = getLinks("/wiki/Kevin_Bacon")
try:
    while len(links) > 0:
         newArticle = links[random.randint(0, len(links)-1)].attrs["href"]
         print(newArticle)
         links = getLinks(newArticle)
finally:
    cur.close()
    conn.close()
```

这里有几点需要注意：首先，`charset='utf8'` 要增加到连接字符串里。这是让连接 `conn` 把所有发送到数据库的信息都当成 UTF-8 编码格式（当然，前提是数据库默认编码已经设置成 UTF-8）。

然后要注意的是 `store` 函数。它有两个参数：`title` 和 `content`，并把这两个参数加到了一个 `INSERT` 语句中并用光标执行，然后用光标进行连接确认。这是一个让光标与连接操作分离的好例子；当光标里存储了一些数据库与数据库上下文（context）的信息时，需要通过连接的确认操作先将信息传进数据库，再将信息插入数据库。

最后要注意的是，`finally` 语句是在程序主循环的外面，代码的最底下。这样做可以保证，无论程序执行过程中如何发生中断或抛出异常（当然，因为网络很复杂，你得随时准备遭遇异常），光标和连接都会在程序结束前立即关闭。无论你是在采集网络，还是处理一个打开连接的数据库，用 `try...finally` 都是一个好主意。

虽然 PyMySQL 规模并不大，但是里面有一些非常实用的函数本书并没有介绍。具体请参考 Python 的 DBAPI 标准文档（http://legacy.python.org/dev/peps/pep-0249/）。

5.3.4 数据库技术与最佳实践

有些人的整个职业生涯都在学习、优化和创造数据库。我不是这类人，这本书也不是那类书。但是，和计算机科学的很多主题一样，有一些技巧你其实可以很快地学会，它们可以让你的数据库变得更高效，让应用的运行速度更快。

首先，给每个数据表都增加一个 id 字段，不会出什么问题。MySQL 里所有的表都至少有一个主键（就是 MySQL 用来排序的字段），因此 MySQL 知道怎么组织主键，通常数据库很难智能地选择主键。究竟是用人造的 id 字段作为主键，还是用那些具有唯一性属性的字段作为主键，比如 username 字段，数据科学家和软件工程师已经争论了很多年，但我更倾向于主动创建一个 id 字段。这样做的原因一两句话难以说清，不过对于一些非企业级系统的数据库，你还是应该用自增的 id 字段作为主键。

其次，用智能索引。字典（指的是常用的工具书，不是指 Python 的字典对象）是按照字母顺序排列的单词表。这样做让你在任何时候都能快速地找到一个单词，只要你知道这个单词是如何拼写的就行。你还可以把字典想象成另一种形式，将单词按照单词含义的字母顺序进行排列。如果你没玩过一些奇怪的游戏，比如危险边缘（Jeopardy）智力游戏，就不能理解游戏的含义，这样的字典就没法用了。但是在数据库查询的工作里，这种按照字段含义进行排序的情况时有发生。比如，你的数据库里可能有一个字段经常要查询：

```
>SELECT * FROM dictionary WHERE definition="A small furry animal that says meow";
+------+-------+-------------------------------------+
| id   | word  | definition                          |
+------+-------+-------------------------------------+
|  200 | cat   | A small furry animal that says meow |
+------+-------+-------------------------------------+
1 row in set (0.00 sec)
```

你可能非常想给这个表增加一个额外的键（除了已经存在的主键 id 之外），让查询变得更快。但是，增加额外的索引需要占用更多的空间，而且插入新行的时候也需要花费更多的时间。为了让事情简单点儿，你可以让 MySQL 只检索查询列的一部分字符。比如下面的命令创建了一个查询 definition 字段前 16 个字符的智能索引：

```
CREATE INDEX definition ON dictionary (id, definition(16));
```

这个索引比全文查询的速度要快很多，而且不需要占用过多的空间和处理时间。

最后一点是关于数据查询时间和数据库空间的问题。一个常见的误区就是在数据库中存储大量重复数据，尤其是在做大量自然语言数据的网络数据采集任务时。举个例子，假如你想统计网站突然出现的一些词组的频率。这些词组也许可以从一个现成的列表里获得，也许可以通过文本分析算法自动提取。最终你可能会把词组储存成下表的形式：

```
+--------+-------------+------+-----+---------+----------------+
| Field  | Type        | Null | Key | Default | Extra          |
```

```
+--------+-------------+------+-----+---------+----------------+
| id     | int(11)     | NO   | PRI | NULL    | auto_increment |
| url    | varchar(200) | YES  |     | NULL    |                |
| phrase | varchar(200) | YES  |     | NULL    |                |
+--------+-------------+------+-----+---------+----------------+
```

每当你发现一个词组就在数据库中增加一行，同时把 URL 记录下来。但是，如果把这些数据分成三个表，你就可以看到数据库占用的空间会大大降低：

```
>DESCRIBE phrases
+--------+-------------+------+-----+---------+----------------+
| Field  | Type        | Null | Key | Default | Extra          |
+--------+-------------+------+-----+---------+----------------+
| id     | int(11)     | NO   | PRI | NULL    | auto_increment |
| phrase | varchar(200) | YES  |     | NULL    |                |
+--------+-------------+------+-----+---------+----------------+

 >DESCRIBE urls
 +-------+-------------+------+-----+---------+----------------+
| Field | Type        | Null | Key | Default | Extra          |
+-------+-------------+------+-----+---------+----------------+
| id    | int(11)     | NO   | PRI | NULL    | auto_increment |
| url   | varchar(200) | YES  |     | NULL    |                |
+-------+-------------+------+-----+---------+----------------+

>DESCRIBE foundInstances
+-------------+---------+------+-----+---------+----------------+
| Field       | Type    | Null | Key | Default | Extra          |
+-------------+---------+------+-----+---------+----------------+
| id          | int(11) | NO   | PRI | NULL    | auto_increment |
| urlId       | int(11) | YES  |     | NULL    |                |
| phraseId    | int(11) | YES  |     | NULL    |                |
| occurrences | int(11) | YES  |     | NULL    |                |
+-------------+---------+------+-----+---------+----------------+
```

虽然表定义的结构变复杂了，但是新增的字段就是 `id` 字段。它们是整数，不会占用很多空间。另外，每个 URL 和词组都只会储存一次。

除非你安装了第三方包或保存详细的数据库日志，否则你无法掌握数据库里数据增加、更新或删除的具体时间。因此，如果需要对数据可用的空间、变更的频率和变更的重要性进行分析，你应该考虑在数据新增、更新或删除时加一个时间戳。

5.3.5 MySQL里的“六度空间游戏”

在第 3 章，我们介绍过“维基百科六度分隔”问题，其目标是通过一些词条链接寻找两个词条间的联系（即找出一条链接路径，只要点击链接就可以从一个维基词条到另一个维基词条）。

为了解决这个问题，我们不仅需要建立网络爬虫采集网页（之前我们已经做过），还要把

采集的信息以某种形式存储起来，以便后续进行数据分析。

前面介绍过的自增的 `id` 字段、时间戳以及多份数据表在这里都要用到。为了确定最合理的信息存储方式，你需要先想想游戏规则。一个链接可以轻易地把页面 A 连接到页面 B。同样也可以轻易地把页面 B 连接到页面 A，不过这可能是另一条链接。我们可以这样识别一个链接，即“页面 A 存在一个链接，可以连接到页面 B”。也就是 `INSERT INTO links (fromPageId, toPageId) VALUES (A, B);`（其中，“A”和“B”分别表示页面的 ID 号）。

因此需要设计一个带有两张数据表的数据库来分别存储页面和链接，两张表都带有创建时间和独立的 ID 号，代码如下所示：

```
CREATE TABLE `wikipedia`.`pages` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `url` VARCHAR(255) NOT NULL,
  `created` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`));

CREATE TABLE `wikipedia`.`links` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `fromPageId` INT NULL,
  `toPageId` INT NULL,
  `created` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`));
```

注意，这里和前面打印页面标题的爬虫不同，我没有在页面数据表里使用页面标题字段，为什么这么做呢？其实是因为页面标题要在你进入页面后读取内容才能抓到。那么，如果我们想创建一个高效的爬虫来填充这些数据表，那么只存储页面的链接就可以保存词条页面了，甚至不需要访问词条页面。

当然并不是所有网站都具有这个特点，但是维基百科的词条链接和对应的页面标题是可以通过简单的操作进行转换的。例如，http://en.wikipedia.org/wiki/Monty_Python 的后面就是页面标题“Monty Python”。

下面的代码会把“贝肯数”（一个页面与凯文・贝肯词条页面的链接数）不超过 6 的维基百科页面存储起来：

```
from bs4 import BeautifulSoup
import re
import pymysql

conn = pymysql.connect(host='127.0.0.1', unix_socket='/tmp/mysql.sock',user=
                       'root', passwd=None, db='mysql', charset='utf8')
cur = conn.cursor()
cur.execute("USE wikipedia")

def insertPageIfNotExists(url):
    cur.execute("SELECT * FROM pages WHERE url = %s", (url))
```

```
    if cur.rowcount == 0:
        cur.execute("INSERT INTO pages (url) VALUES (%s)", (url))
        conn.commit()
        return cur.lastrowid
    else:
        return cur.fetchone()[0]

def insertLink(fromPageId, toPageId):
    cur.execute("SELECT * FROM links WHERE fromPageId = %s AND toPageId = %s",
                  (int(fromPageId), int(toPageId)))
    if cur.rowcount == 0:
        cur.execute("INSERT INTO links (fromPageId, toPageId) VALUES (%s, %s)",
                    (int(fromPageId), int(toPageId)))
        conn.commit()

pages = set()
def getLinks(pageUrl, recursionLevel):
    global pages
    if recursionLevel > 4:
        return;
    pageId = insertPageIfNotExists(pageUrl)
    html = urlopen("http://en.wikipedia.org"+pageUrl)
    bsObj = BeautifulSoup(html)
    for link in bsObj.findAll("a",
                              href=re.compile("^(/wiki/)((?!:).)*$")):
                              insertLink(pageId,
                                     insertPageIfNotExists(link.attrs['href']))
        if link.attrs['href'] not in pages:
            # 遇到一个新页面,加入集合并搜索里面的词条链接
            newPage = link.attrs['href']
            pages.add(newPage)
            getLinks(newPage, recursionLevel+1)
getLinks("/wiki/Kevin_Bacon", 0)
cur.close()
conn.close()
```

用递归实现那些需要运行很长时间的代码，通常是一件复杂的事情。在本例中，变量 `recursionLevel` 被传递到 `getLinks` 函数里，用来跟踪函数递归的次数（每完成一次递归，`recursionLevel` 就加 1）。当 `recursionLevel` 值到 5 的时候，函数会自动返回，不会继续递归。这个限制可以防止数据太大导致内存堆栈溢出。

需要注意的是，这个程序可能要运行好几天才会结束。虽然我自己运行过它，但是我的数据库里只保持了一点点贝肯数不超过 6 的词条，因为维基百科服务器会拒绝程序的请求。但是，这些数据对后面分析维基百科词条的链接路径问题已经足够了。

关于这个问题的补充和最终答案，将在第 8 章关于有向图的问题里介绍。

5.4 Email

与网页通过 HTTP 协议传输一样，邮件是通过 SMTP（Simple Mail Transfer Protocol，简

单邮件传输协议）传输的。而且，和你用网络服务器的客户端（浏览器）处理那些通过 HTTP 协议传输的网页一样，Email 服务器也有客户端，像 Sendmail、Postfix 和 Mailman 等，都可以收发邮件。

虽然用 Python 发邮件很容易，但是需要你连接那些正在运行 SMTP 协议的服务器。在服务器或本地机器上设置 SMTP 客户端有点儿复杂，也超出了本书的介绍范围，但是有很多资料可以帮你解决问题，如果你用的是 Linux 或 Mac OS X 系统，参考资料会更丰富。

下面的代码运行的前提是你的电脑已经可以正常地运行一个 SMTP 客户端。（如果要调整代码用于远程 SMTP 客户端，请把 `localhost` 改成远程服务器地址。）

用 Python 发一封邮件只要 9 行代码：

```
import smtplib
from email.mime.text import MIMEText

msg = MIMEText("The body of the email is here")

msg['Subject'] = "An Email Alert"
msg['From'] = "ryan@pythonscraping.com"
msg['To'] = "webmaster@pythonscraping.com"

s = smtplib.SMTP('localhost')
s.send_message(msg)
s.quit()
```

Python 有两个包可以发送邮件：`smtplib` 和 `email`。

Python 的 `email` 模块里包含了许多实用的邮件格式设置函数，可以用来创建邮件“包裹”。下面的示例中使用的 `MIMEText` 对象，为底层的 MIME（Multipurpose Internet Mail Extensions，多用途互联网邮件扩展类型）协议传输创建了一封空邮件，最后通过高层的 SMTP 协议发送出去。`MIMEText` 对象 `msg` 包括收发邮箱地址、邮件正文和主题，Python 通过它就可以创建一封格式正确的邮件。

`smtplib` 模块用来设置服务器连接的相关信息。就像 MySQL 服务器的连接一样，这个连接必须在用完之后及时关闭，以避免同时创建太多连接而浪费资源。

把这个简单的邮件程序封装成函数后，可以更方便地扩展和使用：

```
import smtplib
from email.mime.text import MIMEText
from bs4 import BeautifulSoup
from urllib.request import urlopen
import time

def sendMail(subject, body):
    msg = MIMEText(body)
```

```
        msg['Subject'] = subject
        msg['From'] = "christmas_alerts@pythonscraping.com"
        msg['To'] = "ryan@pythonscraping.com"

    s = smtplib.SMTP('localhost')
    s.send_message(msg)
    s.quit()

    bsObj = BeautifulSoup(urlopen("https://isitchristmas.com/"))
    while(bsObj.find("a", {"id":"answer"}).attrs['title'] == "NO"):
        print("It is not Christmas yet.")
        time.sleep(3600)
    bsObj = BeautifulSoup(urlopen("https://isitchristmas.com/"))
    sendMail("It's Christmas!",
             "According to http://itischristmas.com, it is Christmas!")
```

这个程序每小时检查一次 https://isitchristmas.com/ 网站（根据日期判断当天是不是圣诞节）。如果页面上的信息不是“NO”（中国用户在网站页面上看到的“NO”在源代码里是 `<noscript>` 不是 `</noscript>`），就会给你发一封邮件，告诉你圣诞节到了。

虽然这个程序看起来并没有墙上的挂历有用，但是稍作修改就可以做很多有用的事情。它可以发送网站访问失败、应用测试失败的异常情况，也可以在 Amazon 网站上出现了一款卖到断货的畅销品时通知你——这些都是挂历做不到的事情。

第 6 章

读取文档

有种观点认为，互联网基本上就是那些符合新式 Web 2.0 潮流，并且经过多媒体内容点缀的 HTML 网站构成的集合，这些内容在网络数据采集时几乎都是要被忽略的。但是，这种观点忽略了互联网最基本的特征：作为不同类型文件的传输媒介。

虽然互联网在 20 世纪 60 年代末期就已经以不同的形式出现，但是 HTML 直到 1992 年才问世。在此之前，互联网基本上就是收发邮件和传输文件；今天看到的网页的概念那时还没有。总之，互联网并不是一个 HTML 页面的集合。它是一个信息集合，而 HTML 文件只是展示信息的一个框架而已。如果我们的爬虫不能读取其他类型的文件，包括纯文本、PDF、图像、视频、邮件等，我们将会失去很大一部分数据。

本章重点介绍文档处理的相关内容，包括把文件下载到文件夹里，以及读取文档并提取数据。我们还会介绍文档的不同编码类型，让程序可以读取非英文的 HTML 页面。

6.1　文档编码

文档编码是一种告诉程序——无论是计算机的操作系统还是 Python 代码——读取文档的规则。文档编码的方式通常可以根据文件的扩展名进行判断，虽然文件扩展名并不是由编码确定的，而是由开发者确定的。例如，如果我把 myImage.jpg 另存为 myImage.txt，不会出现任何问题，但当我用文本编辑器打开它的时候就有问题了。好在这种情况很少见，如果要正确地读取一个文档，必须要知道它的扩展名。

从最底层的角度看，所有文档都是由 0 和 1 编码而成的。而在高层（贴近用户的层级），编码算法会定义“每个字符多少位”或“每个像素的颜色值用多少位”（图像文件里）之

类的事情，在那里你会遇到一些数据压缩算法或体积缩减算法，比如 PNG 图像编码格式（一种无损压缩的位图图形格式）。

虽然第一次处理这些非 HTML 格式的文件时会觉得很没底，但是只要安装了合适的库，Python 就可以帮你处理任意类型的文档。纯文本文件、视频文件和图像文件的唯一区别，就是它们的 0 和 1 面向用户的转换方式不同。在本章后面的内容里，我会介绍几种常用的文档格式：纯文本、PDF、PNG 和 GIF。

6.2 纯文本

虽然把文件存储为在线的纯文本格式并不常见，但是一些简易网站，或者拥有大量纯文本文件的“旧式学术”（old-school）网站经常会这么做。例如，互联网工程任务组（Internet Engineering Task Force，IETF）网站就存储了 IETF 发表过的所有文档，包含 HTML、PDF 和纯文本格式（例如 https://www.ietf.org/rfc/rfc1149.txt）。大多数浏览器都可以很好地显示纯文本文件，采集它们也不会遇到什么问题。

对大多数简单的纯文本文件，像 http://www.pythonscraping.com/pages/warandpeace/chapter1.txt 这个练习文件，你可以用下面的方法读取：

```
from urllib.request import urlopen
textPage = urlopen(
              "http://www.pythonscraping.com/pages/warandpeace/chapter1.txt")
print(textPage.read())
```

通常，当用 `urlopen` 获取了网页之后，我们会把它转变成 BeautifulSoup 对象，方便后面对 HTML 进行分析。在这段代码中，我们直接读取页面内容。你可能觉得，如果把它转变成 BeautifulSoup 对象应该也不错，但那样做其实适得其反——这个页面不是 HTML，所以 BeautifulSoup 库就没用了。一旦纯文本文件被读成字符串，你就只能用普通 Python 字符串的方法分析它了。当然，这么做有个缺点，就是你不能对字符串使用 HTML 标签，去定位那些你真正需要的文字，避开那些你不需要的文字。如果现在你想从纯文本文件中抽取某些信息，还是有些难度的。

文本编码和全球互联网

记得前面我说过，如果你想正确地读取一个文件，知道它的扩展名就可以了。不过非常奇怪的是，这条规则不能应用到最基本的文档格式：.txt 文件。

大多数时候用前面的方法读取纯文本文件都没问题。但是，互联网上的文本文件会比较复杂。下面介绍一些英文和非英文编码的基础知识，包括 ASCII、Unicode 和 ISO 编码，以及对应的处理方法。

1. 编码类型简介

20 世纪 90 年代初，一个叫 Unicode 联盟（The Unicode Consortium）的非营利组织尝试将地球上所有用于书写的符号进行统一编码。其目标包括拉丁字母、斯拉夫字母（кириллица）、中国象形文字（象形）、数学和逻辑符号（∑，≥），甚至表情和“杂项”（miscellaneous）符号，如生化危机标记（☣）和和平符号（☮）等。

编码的结果就是你熟知的 UTF-8，全称是“Universal Character Set - Transformation Format 8 bit”，即“统一字符集－转换格式 8 位”。一个常见的误解是 UTF-8 把所有字符都存储成 8 位。其实“8 位”只是显示一个字符需要的最小位数，而不是最大位数。（如果 UTF-8 的每个字符都是 8 位，那一共也只能存储 2^8 个，即 256 个字符。这对中文字符和其他符号来说显然不够。）

真实情况是，UTF-8 的每个字符开头有一个标记表示“这个字符只用一个字节”或“那个字符需要用两个字节”，一个字符最多可以是四个字节。由于这四个字节里还包含一部分设置信息，用来决定多少字节用做字符编码，所以全部的 32 位（32 位 =4 字节 ×8 位 / 字节）并不会都用，其实最多使用 21 位，也就是总共 2 097 152 种可能里面可以有 1 114 112 个字符。

虽然对很多程序来说，Unicode 都是上帝的礼物（godsend），但是有些习惯很难改变，ASCII 依然是许多英文用户的不二选择。

ASCII 是 20 世纪 60 年代开始使用的文字编码标准，每个字符 7 位，一共 2^7，即 128 个字符。这对于拉丁字母（包括大小写）、标点符号和英文键盘上的所有符号，都是够用的。

在 20 世纪 60 年代，存储的文件用 7 位编码和用 8 位编码之间的差异是巨大的，因为内存非常昂贵。当时，计算机科学家们为了是需要增加一位来获得一个漂亮的二进制数（用 8 位），还是让文件用更少的位数（用 7 位）费尽心机。最终，7 位编码胜利了。但是，在新式的计算方式中，每个 7 位码的前面都补充（pad）了一个“0”[1]，留给我们两个最坏的结果是，文件大了 14%（编码由 7 位变成 8 位，体积增加了 14%），并且由于只有 128 个字符，缺乏灵活性。

在 UTF-8 设计过程中，设计师决定利用 ASCII 文档里的“填充位”，让所有以“0”开头的字节表示这个字符只用 1 个字节，从而把 ASCII 和 UTF-8 编码完美地结合在一起。因此，下面的字符在 ASCII 和 UTF-8 两种编码方式中都是有效的：

```
01000001 - A
01000010 - B
01000011 - C
```

注 1：padding（填充）位在稍后介绍的 ISO 编码标准里还会介绍。

而下面的字符只在 UTF-8 编码里有效，如果文档用 ASCII 编码，那么就会被看成是“无法打印”：

```
11000011 10000000 - À
11000011 10011111 - ß
11000011 10100111 - ç
```

除了 UTF-8，还有其他 UTF 标准，像 UTF-16、UTF-24、UTF-32，不过很少用这些编码标准对文件进行编码，只在一些超出本书介绍范围的环境里使用。

显然，Unicode 标准也有问题，就是任何一种非英文语言文档的体积都比 ASCII 编码的体积大。虽然你的语言可能只需要用大约 100 个字符，像英文的 ASCII 编码，8 位就够了，但是因为是用 UTF-8 编码，所以你还是得用至少 16 位表示每个字符。这会让非英文的纯文本文档体积差不多达到英文文档的两倍，对那些不用拉丁字符集的语言来说都是如此。

ISO 标准解决这个问题的办法是为每种语言创建一种编码。和 Unicode 不同，它使用了与 ASCII 相同的编码，但是在每个字符的开头用 0 作“填充位”，这样就可以让语言在需要的时候创建特殊字符。这种做法对欧洲那些依赖拉丁文字母的语言（编码还是按照 0-127 一一对应）非常合适，只不过需要增加一些特殊字符。这使得 ISO-8859-1（为拉丁文字母设计的）标准里有了分数符号（如 ½）和版权标记符号（©）。

还有一些 ISO 字符集，像 ISO-8859-9（土耳其语）、ISO-8859-2（德语等语言）、ISO-8859-15（法语等语言）也是用类似的规律做出来的。

虽然这些年 ISO 编码标准的使用率一直在下降，但是目前仍有约 9% 的网站使用 ISO 编码[2]，所以有必要做基本的了解，并在采集网站之前需要检查是否使用了这种编码方法。

2. 编码进行时

在上一节里，我们用默认设置的 `urlopen` 读取了网上的 .txt 文档。这么做对英文文档没有任何问题。但是，如果你遇到的是俄语、阿拉伯语文档，或者文档里有一个像“résumé”这样的单词，就可能出问题。

看看下面的代码：

```
from urllib.request import urlopen
textPage = urlopen(
            "http://www.pythonscraping.com/pages/warandpeace/chapter1-ru.txt")
print(textPage.read())
```

这段代码会把《战争与和平》原著（托尔斯泰用俄语和法语写的）的第 1 章打印到屏幕上。打印结果一开头是这样：

注 2：数据源自 http://w3techs.com/technologies/history_overview/character_encoding，通过网络爬虫收集。

```
b"\xd0\xa7\xd0\x90\xd0\xa1\xd0\xa2\xd0\xac \xd0\x9f\xd0\x95\xd0\xa0\xd0\x92\xd0\
x90\xd0\xaf\n\nI\n\n\xe2\x80\x94 Eh bien, mon prince.
```

另外，在大多数浏览器里访问页面也会呈现乱码（如图 6-1）。

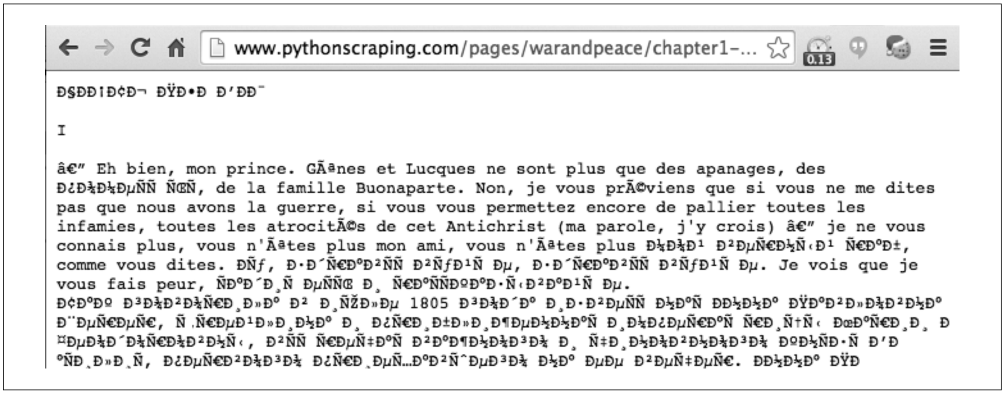


图 6-1：法语和斯拉夫语用浏览器常用的文本编码格式 ISO-8859-1 编码的效果

就算让懂俄语的人来看，这些乱码也难以辨认。这个问题是因为 Python 默认把文本读成 ASCII 编码格式，而浏览器把文本读成 ISO-8859-1 编码格式。其实都不对，应该用 UTF-8 编码格式。

我们可以把字符串显示转换成 UTF-8 格式，这样就可以正确显示斯拉夫文字了：

```
from urllib.request import urlopen

textPage = urlopen(
             "http://www.pythonscraping.com/pages/warandpeace/chapter1-ru.txt")
print(str(textPage.read(), 'utf-8'))
```

用 BeautifulSoup 和 Python 3.x 对文档进行 UTF-8 编码，如下所示：

```
html = urlopen("http://en.wikipedia.org/wiki/Python_(programming_language)")
bsObj = BeautifulSoup(html)
content = bsObj.find("div", {"id":"mw-content-text"}).get_text()
content = bytes(content, "UTF-8")
content = content.decode("UTF-8")
```

你可能打算以后用网络爬虫的时候全部采用 UTF-8 编码读取内容，毕竟 UTF-8 也可以完美地处理 ASCII 编码。但是，要记住还有 9% 的网站使用 ISO 编码格式。所以在处理纯文本文档时，想用一种编码搞定所有的文档依旧不可能。有一些库可以检查文档的编码，或是对文档编码进行估计（用一些逻辑判断“Ñ € Ð°ÑÑÐºÐ°Ð·Ñ”不是单词），不过效果并不是很好。

处理 HTML 页面的时候，网站其实会在 <head> 部分显示页面使用的编码格式。大多数网站，尤其是英文网站，都会带这样的标签：

```
<meta charset="utf-8" />
```

而 ECMA（European Computer Manufacturers Association，欧洲计算机制造商协会，http://www.ecma-international.org/）网站的标签是这样[3]：

```
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-8859-1">
```

如果你要做很多网络数据采集工作，尤其是面对国际网站时，建议你先看看 meta 标签的内容，用网站推荐的编码方式读取页面内容。

6.3 CSV

进行网页采集的时候，你可能会遇到 CSV 文件，也可能有同事希望将数据保存为 CSV 格式。Python 有一个超赞的标准库（https://docs.python.org/3.4/library/csv.html）可以读写 CSV 文件。虽然这个库可以处理各种 CSV 文件，但是这里我重点介绍标准 CSV 格式。如果你在处理 CSV 时有特殊需求，请查看文档！

读取CSV文件

Python 的 csv 库主要是面向本地文件，就是说你的 CSV 文件得存储在你的电脑上。而进行网络数据采集的时候，很多文件都是在线的。不过有一些方法可以解决这个问题：

- 手动把 CSV 文件下载到本机，然后用 Python 定位文件位置；
- 写 Python 程序下载文件，读取之后再把源文件删除；
- 从网上直接把文件读成一个字符串，然后转换成一个 StringIO 对象，使它具有文件的属性。

虽然前两个方法也可以用，但是既然你可以轻易地把 CSV 文件保存在内存里，就不要再下载到本地占硬盘空间了。直接把文件读成字符串，然后封装成 StringIO 对象，让 Python 把它当作文件来处理，就不需要先保存成文件了。下面的程序就是从网上获取一个 CSV 文件（这里用的是 http://pythonscraping.com/files/MontyPythonAlbums.csv 里的 Monty Python 乐团的专辑列表），然后把每一行都打印到命令行里：

```
from urllib.request import urlopen
from io import StringIO
import csv
```

注 3：ECMA 是 ISO 编码标准的主要贡献者之一，所以它的网站用 ISO 编码一点儿也不奇怪。

```
data = urlopen("http://pythonscraping.com/files/MontyPythonAlbums.csv")
              .read().decode('ascii', 'ignore')
dataFile = StringIO(data)
csvReader = csv.reader(dataFile)

for row in csvReader:
    print(row)
```

显示结果很长，开始部分是这样：

```
['Name', 'Year']
["Monty Python's Flying Circus", '1970']
['Another Monty Python Record', '1971']
["Monty Python's Previous Record", '1972']
...
```

从代码中你会发现 `csv.reader` 返回的 `csvReader` 对象是可迭代的，而且由 Python 的列表对象构成。因此，`csvReader` 对象可以用下面的方式接入：

```
for row in csvReader:
    print("The album \""+row[0]+"\" was released in "+str(row[1]))
```

输出结果是：

```
The album "Name" was released in Year
The album "Monty Python's Flying Circus" was released in 1970
The album "Another Monty Python Record" was released in 1971
The album "Monty Python's Previous Record" was released in 1972
...
```

注意看第一行的内容，`The album "Name" was released in Year`。虽然写示例代码的时候，这行内容是否显示都无所谓，但是工作中你肯定不希望将这行信息保留在数据里。有些程序员可能会简单地跳过 `csvReader` 对象的第一行，或者写一个简单的条件把第一行处理掉。不过，还有一个函数可以很好地处理这个问题，那就是 `csv.DictReader`：

```
from urllib.request import urlopen
from io import StringIO
import csv

data = urlopen("http://pythonscraping.com/files/MontyPythonAlbums.csv")
              .read().decode('ascii', 'ignore')
dataFile = StringIO(data)
dictReader = csv.DictReader(dataFile)

print(dictReader.fieldnames)

for row in dictReader:
    print(row)
```

`csv.DictReader` 会返回把 CSV 文件每一行转换成 Python 的字典对象返回，而不是列表对象，并把字段列表保存在变量 `dictReader.fieldnames` 里，字段列表同时作为字典对象的键：

```
['Name', 'Year']
{'Name': "Monty Python's Flying Circus", 'Year': '1970'}
{'Name': 'Another Monty Python Record', 'Year': '1971'}
{'Name': "Monty Python's Previous Record", 'Year': '1972'}
```

虽然用 `DictReaders` 创建、处理和打印 CSV 信息，比 `csvReaders` 要多写一点儿代码，但是考虑到它的便利性和实用性，多写那点儿代码还是值得的。

6.4 PDF

做为一名 Linux 用户，我能理解电脑上没有微软软件却收到了一个 .docx 文件的痛苦，还有就是费半天劲儿找一种能够读取苹果系统媒体文件的解码器。从某种意义上说，Adobe 在 1993 年发明 PDF 格式（Portable Document Format，便携式文档格式）是一种技术革命。PDF 可以让用户在不同的系统上用同样的方式查看图片和文本文档，无论这些文件是在哪种系统上制作的。

虽然把 PDF 显示在网页上已经有点儿过时了（你已经可以把内容显示成 HTML 了，为什么还要用这种静态、加载速度超慢的格式呢？），但是 PDF 仍然无处不在，尤其是在处理商务报表和表单的时候。

2009 年，一个叫 Nick Innes 的英国人上了新闻，他根据英联邦的信息自由法案，要求英国白金汉郡议会公开学生的考试成绩。在几次请求遭到拒绝之后，他开始自己收集信息——最后收集了 184 份 PDF 文件。

虽然 Innes 努力坚持，并且最后得到了一个格式更好的数据库，但是如果他事先了解网络爬虫，再用 Python 众多 PDF 解析模块中的任意一个来处理这些 PDF 文件，那么他一定可以在法庭上节省很多时间。

不过目前很多 PDF 解析库都是用 Python 2.x 版本建立的，还没有迁移到 Python 3.x 版本。但是，因为 PDF 比较简单，而且是开源的文档格式，所以有一些给力的 Python 库可以读取 PDF 文件，而且支持 Python 3.x 版本。

PDFMiner3K 就是一个非常好用的库（是 PDFMiner 的 Python 3.x 移植版）。它非常灵活，可以通过命令行使用，也可以整合到代码中。它还可以处理不同的语言编码，而且对网络文件的处理也非常方便。

你可以下载这个模块的源文件（https://pypi.python.org/pypi/pdfminer3k），解压并用下面命令安装：

```
$python setup.py install
```

文档位于源文件解压文件夹的 /pdfminer3k-1.3.0/docs/index.html 里，这个文档更多是在介

绍命令行接口，而不是 Python 代码整合。

下面的例子可以把任意 PDF 读成字符串，然后用 `StringIO` 转换成文件对象：

```
from urllib.request import urlopen
from pdfminer.pdfinterp import PDFResourceManager, process_pdf
from pdfminer.converter import TextConverter
from pdfminer.layout import LAParams
from io import StringIO
from io import open

def readPDF(pdfFile):
    rsrcmgr = PDFResourceManager()
    retstr = StringIO()
    laparams = LAParams()
    device = TextConverter(rsrcmgr, retstr, laparams=laparams)

    process_pdf(rsrcmgr, device, pdfFile)
    device.close()

    content = retstr.getvalue()
    retstr.close()
    return content

pdfFile = urlopen("http://pythonscraping.com/pages/warandpeace/chapter1.pdf")
outputString = readPDF(pdfFile)
print(outputString)
pdfFile.close()
```

`readPDF` 函数最大的好处是，如果你的 PDF 文件在电脑里，你就可以直接把 `urlopen` 返回的对象 `pdfFile` 替换成普通的 `open()` 文件对象：

```
pdfFile = open("../pages/warandpeace/chapter1.pdf", 'rb')
```

输出结果可能不是很完美，尤其是当 PDF 里有图片、各种各样的文本格式，或者带有表格和数据图的时候。但是，对大多数只包含纯文本内容的 PDF 而言，其输出结果与纯文本格式基本没什么区别。

6.5 微软Word和.docx

冒着被微软朋友鄙视的风险说句话：我不喜欢微软的 Word 软件。并不是因为它是一款烂软件，而且因为它的用户误用了它（好像 Linus Torvalds 对 C++ 的吐槽）。Word 的特异功能就是把那些应该写成简单的 TXT 或 PDF 格式的文件，变成了既大又慢且难以打开的怪兽，它们经常在系统切换和版本切换中出现格式不兼容，而且因为某些原因在文件内容已经定稿后仍处于可编辑的状态。Word 文件从未打算让人频繁传递。不过它们在一些网站上很流行，包括重要的文档、信息，甚至图表和多媒体；总之，那些内容都应该用 HTML 代替。

大约在 2008 年以前，微软 Office 产品中 Word 用 .doc 文件格式。这种二进制格式很难读取，而且能够读取 word 格式的软件很少。为了跟上时代，让自己的软件能够符合主流软件的标准，微软决定使用 Open Office 的类 XML 格式标准，此后新版 Word 文件才与其他文字处理软件兼容，这个格式就是 .docx。

不过，Python 对这种 Google Docs、Open Office 和 Microsoft Office 都在使用的 .docx 格式的支持还不够好。虽然有一个 python-docx 库（http://python-docx.readthedocs.org/en/latest/），但是只支持创建新文档和读取一些基本的文件数据，如文件大小和文件标题，不支持正文读取。如果想读取 Microsoft Office 文件的正文内容，我们需要自己动手找方法。

第一步是从文件读取 XML：

```
from zipfile import ZipFile
from urllib.request import urlopen
from io import BytesIO

wordFile = urlopen("http://pythonscraping.com/pages/AWordDocument.docx").read()
wordFile = BytesIO(wordFile)
document = ZipFile(wordFile)
xml_content = document.read('word/document.xml')
print(xml_content.decode('utf-8'))
```

这段代码把一个远程 Word 文档读成一个二进制文件对象（`BytesIO` 与本章之前用的 `StringIO` 类似），再用 Python 的标准库 `zipfile` 解压（所有的 .docx 文件为了节省空间都进行过压缩），然后读取这个解压文件，就变成 XML 了。

这个 Word 文档在 http://pythonscraping.com/pages/AWordDocument.docx，内容如图 6-2 所示。

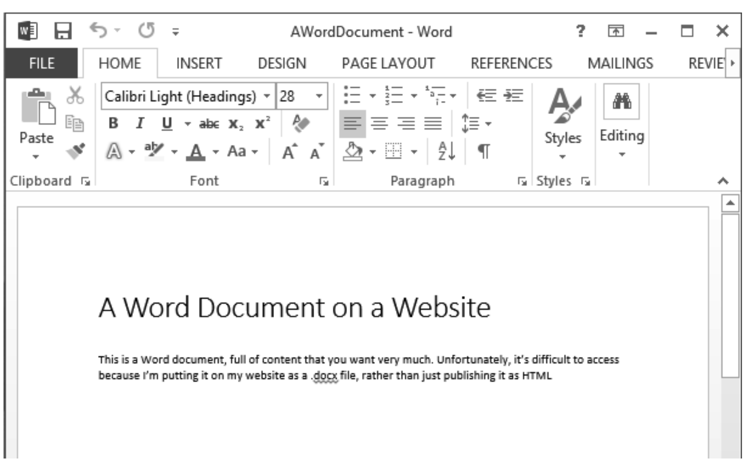


图 6-2：这个 Word 文档的正文内容你可能很想要，但是很难获取，因为我把它放在网站的 .docx 文件里而不是 HTML 里

上面这段 Python 程序读取这个简单的 Word 文档后，输出的结果是这样：

```
<!--?xml version="1.0" encoding="UTF-8" standalone="yes"?-->
<w:document mc:ignorable="w14 w15 wp14" xmlns:m="http://schemas.openx
mlformats.org/officeDocument/2006/math" xmlns:mc="http://schemas.open
xmlformats.org/markup-compatibility/2006" xmlns:o="urn:schemas-micros
oft-com:office:office" xmlns:r="http://schemas.openxmlformats.org/off
iceDocument/2006/relationships" xmlns:v="urn:schemas-microsoft-com:vm
l" xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/m
ain" xmlns:w10="urn:schemas-microsoft-com:office:word" xmlns:w14="htt
p://schemas.microsoft.com/office/word/2010/wordml" xmlns:w15="http://
schemas.microsoft.com/office/word/2012/wordml" xmlns:wne="http://sche
mas.microsoft.com/office/word/2006/wordml" xmlns:wp="http://schemas.o
penxmlformats.org/drawingml/2006/wordprocessingDrawing" xmlns:wp14="h
ttp://schemas.microsoft.com/office/word/2010/wordprocessingDrawing" x
mlns:wpc="http://schemas.microsoft.com/office/word/2010/wordprocessin
gCanvas" xmlns:wpg="http://schemas.microsoft.com/office/word/2010/wor
dprocessingGroup" xmlns:wpi="http://schemas.microsoft.com/office/word
/2010/wordprocessingInk" xmlns:wps="http://schemas.microsoft.com/offi
ce/word/2010/wordprocessingShape"><w:body><w:p w:rsidp="00764658" w:r
sidr="00764658" w:rsidrdefault="00764658"><w:ppr><w:pstyle w:val="Tit
le"></w:pstyle></w:ppr><w:r><w:t>A Word Document on a Website</w:t></
w:r><w:bookmarkstart w:id="0" w:name="_GoBack"></w:bookmarkstart><w:b
ookmarkend w:id="0"></w:bookmarkend></w:p><w:p w:rsidp="00764658" w:r
sidr="00764658" w:rsidrdefault="00764658"></w:p><w:p w:rsidp="0076465
8" w:rsidr="00764658" w:rsidrdefault="00764658" w:rsidrpr="00764658">
<w: r> <w:t>This is a Word document, full of content that you want ve
ry much. Unfortunately, it’s difficult to access because I’m putting
it on my website as a .</w:t></w:r><w:prooferr w:type="spellStart"></
w:prooferr><w:r><w:t>docx</w:t></w:r><w:prooferr w:type="spellEnd"></
w:prooferr> <w:r> <w:t xml:space="preserve"> file, rather than just p
ublishing it as HTML</w:t> </w:r> </w:p> <w:sectpr w:rsidr="00764658"
w:rsidrpr="00764658"> <w:pgszw:h="15840" w:w="12240"></w:pgsz><w:pgm
ar w:bottom="1440" w:footer="720" w:gutter="0" w:header="720" w:left=
"1440" w:right="1440" w:top="1440"></w:pgmar> <w:cols w:space="720"><
/w:cols&g; <w:docgrid w:linepitch="360"></w:docgrid> </w:sectpr> </w:
body> </w:document>
```

确实包含了大量信息，但是被隐藏在 XML 里面。好在文档的所有正文内容都包含在 <w:t> 标签里面，标题内容也是如此，这样就容易处理了。

```
from zipfile import ZipFile
from urllib.request import urlopen
from io import BytesIO
from bs4 import BeautifulSoup

wordFile = urlopen("http://pythonscraping.com/pages/AWordDocument.docx").read()
wordFile = BytesIO(wordFile)
document = ZipFile(wordFile)
xml_content = document.read('word/document.xml')

wordObj = BeautifulSoup(xml_content.decode('utf-8'))
textStrings = wordObj.findAll("w:t")
```

```
    for textElem in textStrings:
        print(textElem.text)
```

这段代码的结果并不完美，但是已经差不多了，一行打印一个 `<w:t>` 标签，可以看到 Word 是如何对文字进行断行处理的：

```
A Word Document on a Website
This is a Word document, full of content that you want very much. Unfortunately,
it's difficult to access because I'm putting it on my website as a . docx
 file, rather than just publishing it as HTML
```

你会看到这里“docx”是单独一行，这是因为在原始的 XML 里，它是由 `<w:proofErr w:type="spellStart"/>` 标签包围的。这是 Word 用红色波浪线高亮显示“docx”的方式，提示这个词可能有拼写错误。

文档的标题是由样式定义标签 `<w:pStyle w:val="Title"/>` 处理的。虽然不能非常简单地定位标题（或其他带样式的文本），但是用 BeautifulSoup 的导航功能还是可以帮助我们解决问题的：

```
textStrings = wordObj.findAll("w:t")
for textElem in textStrings:
    closeTag = ""
    try:
        style = textElem.parent.previousSibling.find("w:pstyle")
        if style is not None and style["w:val"] == "Title":
        print("<h1>")
        closeTag = "</h1>"
    except AttributeError:
    #不打印标签
    pass
    print(textElem.text)
    print(closeTag)
```

这段代码很容易进行扩展，打印不同文本样式的标签，或者把它们标记成其他形式。

第二部分

高级数据采集

你已经掌握了网络数据采集的一些基础知识，现在让我们进入更有趣的第二部分。到目前为止，我们创建的网络爬虫还不是特别给力。如果网络服务器不能立即提供样式规范的信息，爬虫就不能正确地采集数据。如果爬虫只能采集那些显而易见的信息，不经过处理就简单地存储起来，那么迟早要被登录表单、网页交互以及 JavaScript 困住手脚。总之，目前爬虫还没有足够的实力去采集各种数据，只能处理那些愿意被采集的信息。

这部分内容就是要帮你分析原始数据，获取隐藏在数据背后的故事——网站的真实故事其实都隐藏在 JavaScript、登录表单和网站反抓取措施的背后。

通过这部分内容的学习，你将掌握如何用网络爬虫测试网站，自动化处理，以及通过更多的方式接入网络。最后你将学到一些数据采集的工具，帮助你在不同的环境中收集和操作任意类型的网络数据，深入互联网的每个角落。

第 7 章

数据清洗

到目前为止，我们还没有处理过那些样式不规范的数据，要么是使用样式规范的数据源，要么就是彻底放弃样式不符合我们预期的数据。但是在网络数据采集中，你通常无法对采集的数据样式太挑剔。

由于错误的标点符号、大小写字母不一致、断行和拼写错误等问题，零乱的数据（dirty data）是网络中的大问题。本章将介绍一些工具和技术，通过改变代码的编写方式，帮你从源头控制数据零乱的问题，并且对已经进入数据库的数据进行清洗。

7.1　编写代码清洗数据

和写代码处理异常一样，你也应该学习编写预防型代码来处理意外情况。

在语言学里有一个模型叫 n-gram，表示文字或语言中的 *n* 个连续的单词组成的序列。在进行自然语言分析时，使用 n-gram 或者寻找常用词组，可以很容易地把一句话分解成若干个文字片段。

这一节我们将重点介绍如何获取格式合理的 n-gram，并不用它们做任何分析。在第 8 章，我们再用 2-gram 和 3-gram 来做文本摘要提取和语法分析。

下面的代码将返回维基百科词条“Python programming language”的 2-gram 列表：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

def ngrams(input, n):
```

```
  input = input.split(' ')
  output = []
  for i in range(len(input)-n+1):
    output.append(input[i:i+n])
  return output

html = urlopen("http://en.wikipedia.org/wiki/Python_(programming_language)")
bsObj = BeautifulSoup(html)
content = bsObj.find("div", {"id":"mw-content-text"}).get_text()
ngrams = ngrams(content, 2)
print(ngrams)
print("2-grams count is: "+str(len(ngrams)))
```

ngrams 函数把一个待处理的字符串分成单词序列（假设所有单词按照空格分开），然后增加到 n-gram 模型（本例中是 2-gram）里形成以每个单词开始的二元数组。

这段程序会从文字中返回一些有意思同时也很有用的 2-gram 序列：

```
['of', 'free'], ['free', 'and'], ['and', 'open-source'], ['open-source', 'softwa
re']
```

不过，同时也会出现一些零乱的数据：

```
['software\nOutline\nSPDX\n\n\n\n\n\n\n\n\nOperating', 'system\nfamilies\n\n\n\n
AROS\nBSD\nDarwin\neCos\nFreeDOS\nGNU\nHaiku\nInferno\nLinux\nMach\nMINIX\nOpenS
olaris\nPlan'], ['system\nfamilies\n\n\n\nAROS\nBSD\nDarwin\neCos\nFreeDOS\nGNU\
nHaiku\nInferno\nLinux\nMach\nMINIX\nOpenSolaris\nPlan', '9\nReactOS\nTUD:OS\n\n
\n\n\n\n\n\nDevelopment\n\n\n\nBasic'], ['9\nReactOS\nTUD:OS\n\n\n\n\n\n\n\n\n
Development\n\n\n\nBasic', 'For']
```

另外，因为每个单词（除了最后一个单词）都要创建一个 2-gram 序列，所以这个词条里共有 7411 个 2-gram 序列。这并不是一个非常便于管理的数据集！

让我们首先用一些正则表达式来移除转义字符（\n），再把 Unicode 字符过滤掉。我们可以通过下面的函数对之前输出的结果进行清理：

```
def ngrams(input, n):
    content = re.sub('\n+', " ", content)
    content = re.sub(' +', " ", content)
    content = bytes(content, "UTF-8")
    content = content.decode("ascii", "ignore")
    print(content)
    input = input.split(' ')
    output = []
    for i in range(len(input)-n+1):
        output.append(input[i:i+n])
    return output
```

这里首先把内容中的换行符（或者多个换行符）替换成空格，然后把连续的多个空格替换成一个空格，确保所有单词之间只有一个空格。最后，把内容转换成 UTF-8 格式以消除转义字符。

这几步已经可以大大改善输出结果了，但是还有一些问题：

```
['Pythoneers.[43][44]', 'Syntax'], ['7', '/'], ['/', '3'], ['3', '=='], ['==', '
2']
```

因此，需要再增加一些规则来处理数据。我们还可以制定一些规则让数据变得更规范：

- 剔除单字符的“单词”，除非这个字符是“i”或“a”；
- 剔除维基百科的引用标记（方括号包裹的数字，如 [1]）；
- 剔除标点符号（注意：这个规则有点儿矫枉过正，在第 9 章我们将详细介绍，本例暂时这样处理）。

现在“清洗任务”列表变得越来越长，让我们把规则都移出来，单独建一个函数，就叫 `cleanInput`：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
import string

def cleanInput(input):
    input = re.sub('\n+', " ", input)
    input = re.sub('\[[0-9]*\]', "", input)
    input = re.sub(' +', " ", input)
    input = bytes(input, "UTF-8")
    input = input.decode("ascii", "ignore")
    cleanInput = []
    input = input.split(' ')
    for item in input:
        item = item.strip(string.punctuation)
        if len(item) > 1 or (item.lower() == 'a' or item.lower() == 'i'):
            cleanInput.append(item)
    return cleanInput

def ngrams(input, n):
    input = cleanInput(input)
    output = []
    for i in range(len(input)-n+1):
        output.append(input[i:i+n])
    return output
```

这里用 `import string` 和 `string.punctuation` 来获取 Python 所有的标点符号。你可以在 Python 命令行看看标点符号有哪些：

```
>>> import string
>>> print(string.punctuation)
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

在循环体中用 `item.strip(string.punctuation)` 对内容中的所有单词进行清洗，单词两端的任何标点符号都会被去掉，但带连字符的单词（连字符在单词内部）仍然会保留。

这样输出的 2-gram 结果就更干净了：

```
['Linux', 'Foundation'], ['Foundation', 'Mozilla'], ['Mozilla', 'Foundation'], [
'Foundation', 'Open'], ['Open', 'Knowledge'], ['Knowledge', 'Foundation'], ['Fou
ndation', 'Open'], ['Open', 'Source']
```

数据标准化

每个人都会遇到一些样式设计不够人性化的网页，比如“请输入你的电话号码。号码格式必须是 xxx-xxx-xxxx”。

作为一名优秀的程序员，你可能会问：“为什么不自动地对输入的信息进行清洗，去掉非数字内容，然后自动把数据加上分隔符呢？”数据标准化过程要确保清洗后的数据在语言学或逻辑上是等价的，比如电话号码虽然显示成“(555) 123-4567”和“555.123.4567”两种形式，但是实际号码是一样的。

还用之前的 n-gram 示例，让我们在上面增加一些数据标准化特征。

这段代码有一个明显的问题，就是输出结果中包含太多重复的 2-gram 序列。程序把每个 2-gram 序列都加入了列表，没有统计过序列的频率。掌握 2-gram 序列的频率，而不只是知道某个序列是否存在，这不仅很有意思，而且有助于对比不同的数据清洗和数据标准化算法的效果。如果数据标准化成功了，那么唯一的 n-gram 序列数量就会减少，而 n-gram 序列的总数（任何一个 n-gram 序列和与之重复的序列被看成一个 n-gram 序列）不变。也就是说，同样数量的 n-gram 序列，经过去重之后“容量”（bucket）会减少。

不过 Python 的字典是无序的，不能像数组一样直接对 n-gram 序列频率进行排序。字典内部元素的位置不是固定的，排序之后再次使用时还是会变化，除非你把排序过的字典里的值复制到其他类型中进行排序。在 Python 的 `collections` 库里面有一个 `OrderedDict` 可以解决这个问题：

```
from collections import OrderedDict

...

ngrams = ngrams(content, 2)
ngrams = OrderedDict(sorted(ngrams.items(), key=lambda t: t[1], reverse=True))
print(ngrams)
```

这里我用了 Python 的排序函数（https://docs.python.org/3/howto/sorting.html）把序列频率转换成 `OrderedDict` 对象，并按照频率值排序。结果如下：

```
("['Software', 'Foundation']", 40), ("['Python', 'Software']", 38), ("['of', 'th
e']", 35), ("['Foundation', 'Retrieved']", 34), ("['of', 'Python']", 28), ("['in
', 'the']", 21), ("['van', 'Rossum']", 18)
```

在写作本书的时候，词条内容一个有 7696 个 2-gram 序列，其中不重复的 2-gram 序列有 6005 个，频率最高的 2-gram 序列是“Software Foundation”和“Python Software”。但是，仔细观察结果发现还有字母大小写的影响，“Python Software”有两次是“Python software”的形式。同样，“van Rossum”和“Van Rossum”也是作为两个序列统计的。

因此，增加一行代码：

```
input = input.upper()
```

到 `cleanInput` 函数里，这样 2-gram 序列的总数还是 7696，而不重复的 2-gram 序列减少到了 5882 个。

除了这些，还需要再考虑一下，自己计划为数据标准化的进一步深入再投入多少计算能力。很多单词在不同的环境里会使用不同的拼写形式，其实都是等价的，但是为了解决这种等价关系，你需要对每个单词进行检查，判断是否和其他单词有等价关系。

比如，“Python 1st”和“Python first”都出现在 2-gram 序列列表里。但是，如果增加一条规则：“让所有‘first’‘second’‘third’……与 1st，2nd，3rd……等价”，那么每个单词都要额外增加十几次检查。

同理，连字符使用不一致（像“co-ordinated”和“coordinated”）、单词拼写错误以及其他语病（incongruities），都可能对 n-gram 序列的分组结果造成影响，如果语病很严重的话，还可能彻底打乱输出结果。

对带连字符单词的一个处理方法是，先把连字符去掉，然后把单词当作一个字符串，这可能需要在程序中增加一步操作。但是，这样做也可能把带连字符的短语（这是很常见的，比如“just-in-time”“object-oriented”等）处理成一个字符串。要是换一种做法，把连字符替换成空格可能更好一点儿。但是就得准备见到“co ordinated”和“ordinated attack”之类的 2-gram 序列了！

7.2 数据存储后再清洗

对于编写代码清洗数据，你能做或想做的事情只有这些。除此之外，你可能还需要处理一些别人创建的数据库，或者要对一个之前没接触过的数据库进行清洗。

很多程序员遇到这种情况的自然反应就是“写个脚本”，当然这也是一个很好的解决方法。但是，还有一些第三方工具，像 OpenRefine，不仅可以快速简单地清理数据，还可以让非编程人员轻松地看见和使用你的数据。

OpenRefine

OpenRefine（http://openrefine.org/）是 Metaweb 公司 2009 年发布的一个开源软件。

Google 在 2010 年收购了 Metaweb，把项目的名称从 Freebase Gridworks 改成了 Google Refine。2012 年，Google 放弃了对 Refine 的支持，让它重新成为开源软件，名字改成了 OpenRefine，现在每个人都可以为这个项目做贡献。

1. 安装

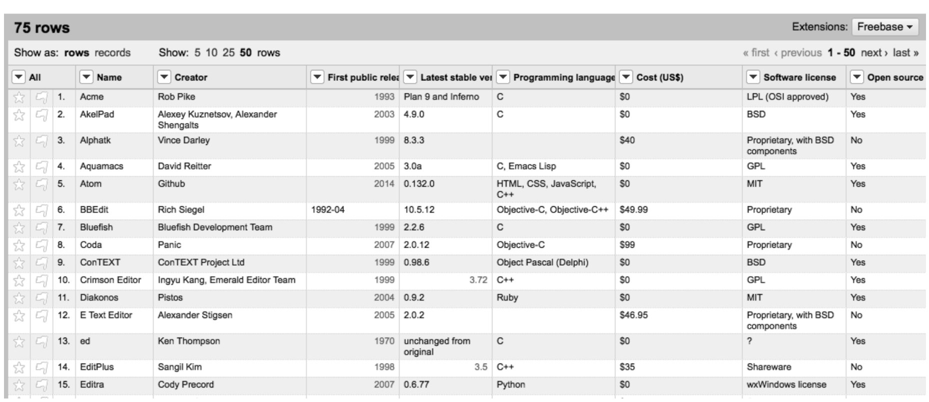
OpenRefine 的独特之处在于虽然它的界面是一个浏览器，但实际上是一个桌面应用，必须下载并安装。你可以从它的下载页面（http://openrefine.org/download.html）下载对应 Linux、Windows 和 Mac OS X 系统的版本。

> 如果你是 Mac 用户，打开安装文件的时候遇到了安装权限问题，请到系统偏好设置→安全性与隐私→通用，把“允许从以下位置下载的应用程序”的选项设置为“任何来源”。从 Google 项目转成开源项目之后，OpenRefine 好像在苹果系统中失去了合法性，不再是来源合法的应用程序了。

要使用 OpenRefine，你得把数据转换成 CSV 文件（如果你需要了解如何生成 CSV 文件，请参考 5.2 节的内容）。另外，如果你的数据已经保存在数据库中，你可能要先把数据导成 CSV 格式。

2. 使用OpenRefine

在下面的例子中，我们将使用维基百科的“文本编辑器对比”表格（https://en.wikipedia.org/wiki/Comparison_of_text_editors）里的内容，如图 7-1 所示。虽然这个表的样式比较规范，但里面包含了多次编辑的痕迹，所以还是有一些样式不一致的地方。另外，因为这个数据是写给人看的，而不是让机器看的，所以原来使用的一些样式（比如用“Free”而不是“$0.00”）不太合适作为 OpenRefine 程序的输入数据。



75 rows — Extensions: Freebase

Show as: rows records — Show: 5 10 25 50 rows — « first ‹ previous 1 - 50 next › last »

All	Name	Creator	First public relea	Latest stable ver	Programming language	Cost (US$)	Software license	Open source
1.	Acme	Rob Pike	1993	Plan 9 and Inferno	C	$0	LPL (OSI approved)	Yes
2.	AkelPad	Alexey Kuznetsov, Alexander Shengalts	2003	4.9.0	C	$0	BSD	Yes
3.	Alphatk	Vince Darley	1999	8.3.3		$40	Proprietary, with BSD components	No
4.	Aquamacs	David Reitter	2005	3.0a	C, Emacs Lisp	$0	GPL	Yes
5.	Atom	Github	2014	0.132.0	HTML, CSS, JavaScript, C++	$0	MIT	Yes
6.	BBEdit	Rich Siegel	1992-04	10.5.12	Objective-C, Objective-C++	$49.99	Proprietary	No
7.	Bluefish	Bluefish Development Team	1999	2.2.6	C	$0	GPL	Yes
8.	Coda	Panic	2007	2.0.12	Objective-C	$99	Proprietary	No
9.	ConTEXT	ConTEXT Project Ltd	1999	0.98.6	Object Pascal (Delphi)	$0	BSD	Yes
10.	Crimson Editor	Ingyu Kang, Emerald Editor Team	1999	3.72	C++	$0	GPL	Yes
11.	Diakonos	Pistos	2004	0.9.2	Ruby	$0	MIT	Yes
12.	E Text Editor	Alexander Stigsen	2005	2.0.2		$46.95	Proprietary, with BSD components	No
13.	ed	Ken Thompson	1970	unchanged from original	C	$0	?	Yes
14.	EditPlus	Sangil Kim	1998	3.5	C++	$35	Shareware	No
15.	Editra	Cody Precord	2007	0.6.77	Python	$0	wxWindows license	Yes

图 7-1：显示在 OpenRefine 屏幕上的维基百科的“文本编辑器对比”表格数据

使用 OpenRefine 会看到每一列的标签旁边都有一个箭头。这个箭头提供了一个工具菜单，可以对这一列数据执行筛选、排序、变换或删除等操作。

筛选。数据筛选可以通过两种方法实现：过滤器（filter）和切片器（facet）。过滤器可以用正则表达式筛选数据，比如“只显示编程语言这一列中用逗号分隔且超过三种编程语言的所有行”，结果如图 7-2 所示。

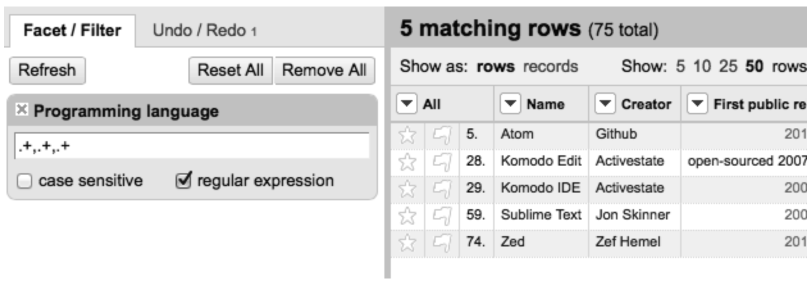


图 7-2：正则表达式“.+,.+,.+”选择用逗号分隔且至少有三种编程语言的所有行

过滤器可以通过右边的操作框轻松地组合、编辑和增加数据，还可以和切片器配合使用。

切片器可以很方便地对一列的部分数据进行包含和不包含的筛选（比如，“显示使用 GPL 和 MIT 授权且在 2005 年之后首次发行的所有行”，结果如图 7-3 所示）。它们都有内置的筛选工具。例如，数值筛选功能会为你提供一个数值滑动窗口，让你选择需要的数值区间。

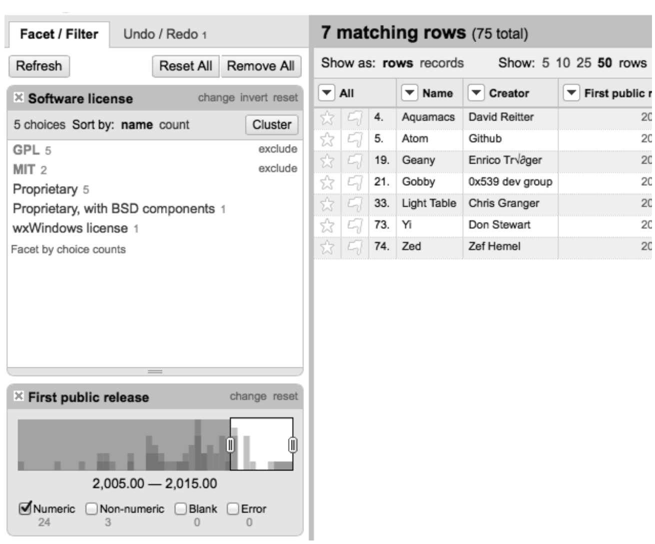


图 7-3：显示使用 GPL 和 MIT 授权且在 2005 年之后首次发行的所有行

经过选过的数据结果可以被导成任意一种 OpenRefine 支持的数据文件格式，包括 CSV、HTML（HTML 表格）、Excel 以及其他格式。

清洗。只有当数据一开始就比较干净时，数据筛选才可以直接快速地完成。例如，在上一节切片器的例子中，有个文本编辑器的发行日期是“01-01-2006”，而真正要寻找的数值是“2006”，所以不能匹配，会被忽略掉，因此在“First public release”切片器中就不会显示了。

OpenRefine 的数据变换功能是通过 OpenRefine 表达式语言（Expression Language）实现的，被称为 GREL（“G”是 OpenRefine 之前的名字 GoogleRefine）。这个语言通过创建规则简单的 lambda 函数来实现数据的转换。例如：

```
if(value.length() != 4, "invalid", value)
```

如果把这个函数应用到“First stable release”列，它就只会保留那些“YYYY”形式的数值，把其他数值标记成“invalid”（无效数据）。

点击列标签旁边的向下箭头，再点击“Edit cells”→“transform”，就可以使用任何 GREL 语句。上面函数的运行结果如图 7-4 所示。

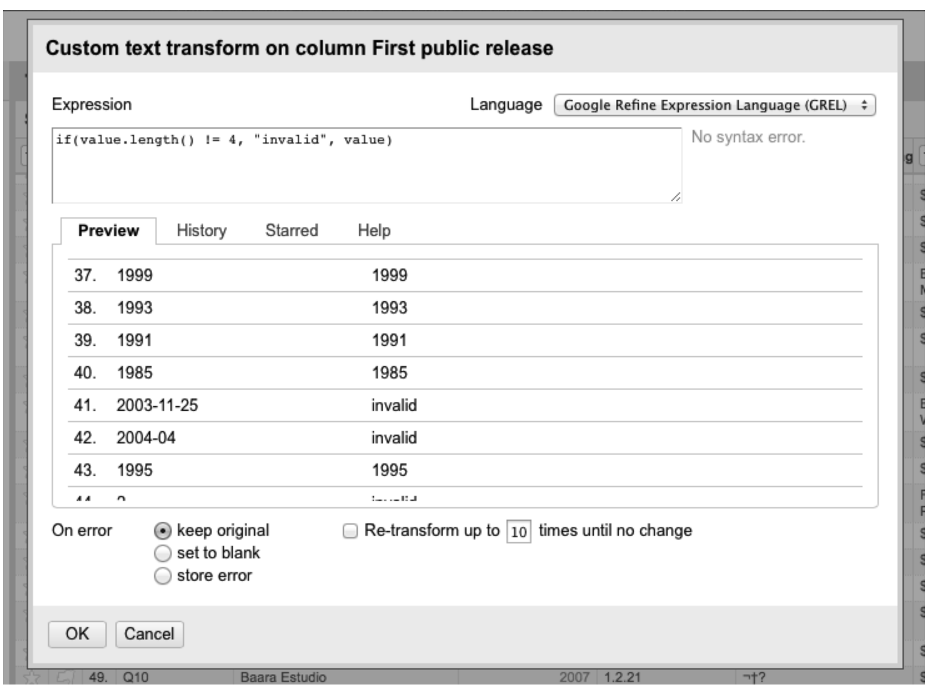


图 7-4：在项目中插入一行 GREL 语句（结果预览显示在语句下面）

但是，把不符合条件的数据标记成无效数据，虽然可以让它们变得容易识别，但是对我们用处不大。我们想要的是尽可能地修复那些格式不规范的数据。这可以用 GERL 的 `match` 函数实现：

```
value.match(".*([0-9]{4}).*").get(0)
```

这是用正则表达式对字符串数据进行匹配。如果正则表达式能够匹配出结果，就会返回一个数组。任何符合正则表达式“捕获组”（capture group）条件的子字符串（指的是括号里的表达式，本例中是“[0-9]{4}”）都会作为数组数值返回。

其实，这行代码会从一个单元格中找出所有的连续的四位整数，然后返回第一个匹配结果。一般情况下，用这个函数从文字或格式不规范的数据中提取年份完全可行。如果正则表达式没有找到年份，就会返回“null”（GERL 操作 null 变量的时候不会抛出空指针异常）。

OpenRefine 还有许多关于单元格编辑和 GERL 数据变换的方法。详细介绍在 OpenRefine 的 GitHub 页面（https://github.com/OpenRefine/OpenRefine/wiki/General-Refine-Expression-Language）。

第 8 章

自然语言处理

到目前为止，我们处理的数据大部分都是数字或数值（countable value）。大多数情况下，我们只是简单地存储数据，没有分析数据。在这一章里，我们将尝试探索英语这个复杂的主题。[1]

当你在 Google 的图片搜索里输入“cute kitten”时，Google 怎么会知道你要搜索什么呢？其实这个词组与可爱的小猫咪是密切相关的。当你在 YouTube 搜索框中输入“dead parrot”的时候，YouTube 怎么会知道要推荐一些 Monty Python 乐团的幽默短剧呢？那是因为每个上传的视频里都带有标题和简介文字。

其实，输入“deceased bird monty python”这类短语时，即使页面上没有单词“deceased”或“bird”，也会立即显示同样的“Dead Parrot”幽默短剧。Google 知道“hot dog”是一种食物，“boiling puppy”却是另一种完全不同的东西。它究竟是怎么实现的呢？其实这一切都是统计学在起作用！

虽然你可能认为自己的项目和文本分析没有任何关系，但是理解文本分析的原理对各种机器学习场景都是非常有用的，而且还可以进一步提高自己利用概率论和算法知识对现实问题进行抽象建模的能力。

注 1：虽然这一章介绍的很多方法可以用于大多数其他语种，但是目前只关注英语的自然语言处理是没有问题的。像 Python 的自然语言处理工具包（Natural Language Toolkit，NLTK）就是面向英语的。互联网上 55% 的内容依然是英文（其次是俄语和德语，均不足 6%，http://w3techs.com/technologies/overview/content_language/all）。但是谁知道未来会怎样呢？英语占互联网大头的情况未来几乎肯定会变化，而且未来几年内就需要改变。

例如，Shazam 音乐雷达是一种可以识别出一段音频中包含哪首歌的服务，即使音频中包含了环绕的噪声或失真也没问题。Google 正在通过图片内容识别自动增加识别文字。[2] 比如对已知的热狗图片和其他热狗图片进行对比，搜索引擎就可以通过不断地学习掌握热狗的特征，然后对其他图片进行模式识别，判断图片是不是热狗。

8.1 概括数据

在第 7 章里，我们介绍过如何把文本内容分解成 n-gram 模型，或者说是 *n* 个单词长度的词组。从最基本的功能上说，这个集合可以用来确定这段文字中最常用的单词和短语。另外，还可以提取原文中那些最常用的短语周围的句子，对原文进行看似合理的概括。

我们即将用来做数据归纳的文字样本源自美国第九任总统威廉 · 亨利 · 哈里森的就职演说。哈里森的总统生涯创下美国总统任职历史的两个记录：一个是最长的就职演说，另一个是最短的任职时间——32 天。

我们将用他的总统就职演说（http://pythonscraping.com/files/inaugurationSpeech.txt）的全文作为这一章许多示例代码的数据源。

简单修改一下我们在第 7 章里用过的 n-gram 模型，就可以获得 2-gram 序列的频率数据，然后我们用 Python 的 `operator` 模块对 2-gram 序列的频率字典进行排序：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import re
import string
import operator

def cleanInput(input):
    input = re.sub('\n+', " ", input).lower()
    input = re.sub('\[[0-9]*\]', "", input)
    input = re.sub(' +', " ", input)
    input = bytes(input, "UTF-8")
    input = input.decode("ascii", "ignore")
    cleanInput = []
    input = input.split(' ')
    for item in input:
        item = item.strip(string.punctuation)
        if len(item) > 1 or (item.lower() == 'a' or item.lower() == 'i'):
            cleanInput.append(item)
    return cleanInput

def ngrams(input, n):
    input = cleanInput(input)
```

注 2：详情请见“A Picture Is Worth a Thousand (Coherent) Words: Building a Natural Description of Images”（一图胜千言：图像含义自动描述的实现方法），2014 年 11 月 17 日（http://bit.ly/1HEJ8kX）。

```
    output = {}
    for i in range(len(input)-n+1):
        ngramTemp = " ".join(input[i:i+n])
        if ngramTemp not in output:
            output[ngramTemp] = 0
        output[ngramTemp] += 1
    return output

content = str(
      urlopen("http://pythonscraping.com/files/inaugurationSpeech.txt").read(),
              'utf-8')
ngrams = ngrams(content, 2)
sortedNGrams = sorted(ngrams.items(), key = operator.itemgetter(1), reverse=True)
print(sortedNGrams)
```

输出结果的一部分是：

```
[('of the', 213), ('in the', 65), ('to the', 61), ('by the', 41), ('t
he constitution', 34), ('of our', 29), ('to be', 26), ('from the', 24
), ('the people', 24), ('and the', 23), ('it is', 23), ('that the', 2
3), ('of a', 22), ('of their', 19)
```

这些 2-gram 序列中，“the constitution”像是演说的主旨，“of the”“in the”和“to the”看起来并不重要。怎么才能用准确的方式去掉这些不想要的词呢？

前人已经仔细地研究过这些“有意义的”单词和“没意义的”单词的差异了，他们的工作可以帮助我们完成过滤工作。美国杨百翰大学的语言学教授 Mark Davies 一直在维护当代美式英语语料库（Corpus of Contemporary American English，http://corpus.byu.edu/coca/），里面包含了 1990~2012 年美国畅销作品里的超过 4.5 亿个单词。

最常用的 5000 个单词列表可以免费获取，作为一个基本的过滤器来过滤最常用的 2-gram 序列绰绰有余。其实只用前 100 个单词就可以大幅改善分析结果，我们增加一个 isCommon 函数来实现：

```
def isCommon(ngram):
    commonWords = ["the", "be", "and", "of", "a", "in", "to", "have", "it",
        "i", "that", "for", "you", "he", "with", "on", "do", "say", "this",
        "they", "is", "an", "at", "but","we", "his", "from", "that", "not",
        "by", "she", "or", "as", "what", "go", "their","can", "who", "get",
        "if", "would", "her", "all", "my", "make", "about", "know", "will",
        "as", "up", "one", "time", "has", "been", "there", "year", "so",
        "think", "when", "which", "them", "some", "me", "people", "take",
        "out", "into", "just", "see", "him", "your", "come", "could", "now",
        "than", "like", "other", "how", "then", "its", "our", "two", "more",
        "these", "want", "way", "look", "first", "also", "new", "because",
        "day", "more", "use", "no", "man", "find", "here", "thing", "give",
        "many", "well"]
    for word in ngram:
        if word in commonWords:
            return True
    return False
```

这样处理之后，就可以得到在样本文字中出现的频率不低于三次的 2-gram 序列，如下所示：

```
('united states', 10), ('executive department', 4), ('general governm
ent', 4), ('called upon', 3), ('government should', 3), ('whole count
ry', 3), ('mr jefferson', 3), ('chief magistrate', 3), ('same causes'
, 3), ('legislative body', 3)
```

效果看着不错，列表中的前两项是"United States"和"executive department"，和我们对就职演说的期待是一样的。

这里需要注意的是，我们是用比较新的常用词列表过滤结果的，这对 1841 年写出来的文字来说可能不是非常合适。但是，因为我们只用了列表里前 100 个单词——我们姑且可以认为，随着年代的变化，这 100 个单词应该比列表最后的 100 个单词要更具稳定性——而且我们也获得了满意的结果，所以好像也不必挖掘或创建一组 1841 年最常用的单词列表（虽然这样的努力可能会很有趣）。

现在一些核心的主题词已经从文本中抽取出来了，它们怎么帮助我们归纳这段文字呢？一种方法是搜索包含每个核心 n-gram 序列的第一句话，这个方法的理论是英语中段落的首句往往是对后面内容的概述。前五个 2-gram 序列的搜索结果是：

- The Constitution of the United States is the instrument containing this grant of power to the several departments composing the government.
- Such a one was afforded by the executive department constituted by the Constitution.
- The general government has seized upon none of the reserved rights of the states.
- Called from a retirement which I had supposed was to continue for the residue of my life to fill the chief executive office of this great and free nation, I appear before you, fellow-citizens, to take the oaths which the constitution prescribes as a necessary qualification for the performance of its duties; and in obedience to a custom coeval with our government and what I believe to be your expectations I proceed to present to you a summary of the principles which will govern me in the discharge of the duties which I shall be called upon to perform.
- The presses in the necessary employment of the government should never be used to clear the guilty or to varnish crime.

当然，这些估计还不能马上发布到 CliffsNotes 上面，但是考虑到全文原来一共有 217 句话，而这里的第四句话（"Called from a retirement..."）已经把主题总结得很好了，作为初稿应该能凑合。

8.2 马尔可夫模型

你可能听说过马尔可夫文字生成器（Markov text generator）。它们是一种非常受欢迎的娱

乐方式，像 Twitov App（http://twitov.extrafuture.com/），还可以用它们为傻瓜测试系统生成看似真实的垃圾邮件。

这些文字生成器都是基于一种常用于分析大量随机事件的马尔可夫模型，随机事件的特点是一个离散事件发生之后，另一个离散事件将在前一个事件的条件下以一定的概率发生。

例如，我们可以对一个天气系统建立马尔可夫模型，如图 8-1 所示。

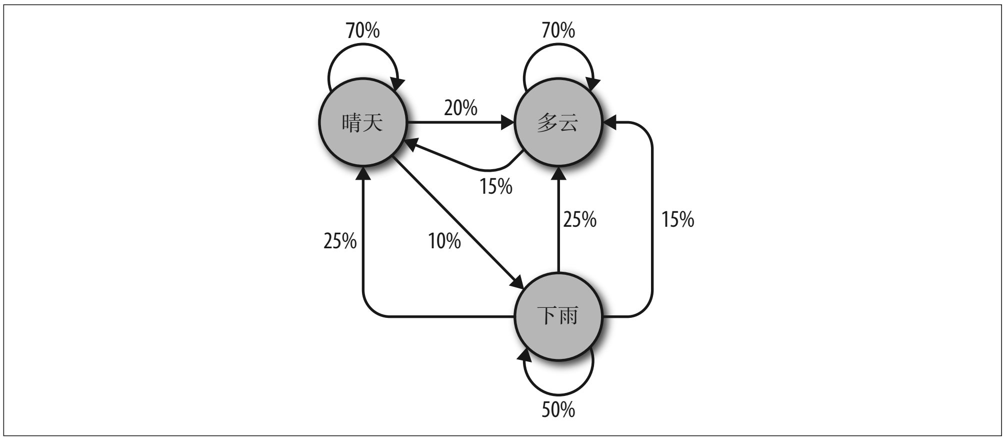


图 8-1：马尔可夫模型描述的理论天气系统

在这个天气系统模型中，如果今天是晴天，那么明天有 70% 的可能是晴天，20% 的可能多云，10% 的可能下雨。如果今天是下雨天，那么明天有 50% 的可能也下雨，25% 的可能是晴天，25% 的可能是多云。

需要注意以下几点。

- 任何一个节点引出的所有可能的总和必须等于 100%。无论是多么复杂的系统，必然会在下一步发生若干事件中的一个事件。
- 虽然这个天气系统在任一时间都只有三种可能，但是你可以用这个模型生成一个天气状态的无限次转移列表。
- 只有当前节点的状态会影响后一天的状态。如果你在“晴天”节点上，即使前 100 天都是晴天或雨天都没关系，明天晴天的概率还是 70%。
- 有些节点可能比其他节点较难到达。这个现象的原因用数学来解释非常复杂，但是可以直观地看出，在这个系统中任意时间节点上，第二天是“雨天”的可能性（指向它的箭头概率之和小于“100%”）比“晴天”或“多云”要小很多。

很明显，这是一个很简单的系统，而马尔可夫模型可以演化成任意规模的复杂系统。事实上，Google 的 page rank 算法也是基于马尔可夫模型，把网站做为节点，入站 / 出站链接做为节点之间的连线。连接某一个节点的“可能性”（likelihood）表示一个网站的相对关

注度。也就是说，如果我们的天气系统表示一个微型互联网，那么“雨天”的页面等级（page rank）相对比较低，而“多云”的页面等级相对比较高。

了解了这些概念之后，让我们再回到本节的主题，研究一个具体的例子：文本分析与写作。

还用前面例子里分析的威廉 · 亨利 · 哈里森的就职演讲内容，我们可以写出下面的代码，通过演讲内容的结构生成任意长度的（下面示例中链长为 100）马尔可夫链组成的句子：

```
from urllib.request import urlopen
from random import randint

def wordListSum(wordList):
    sum = 0
    for word, value in wordList.items():
        sum += value
    return sum

def retrieveRandomWord(wordList):

    randIndex = randint(1, wordListSum(wordList))
    for word, value in wordList.items():
        randIndex -= value
        if randIndex <= 0:
            return word

def buildWordDict(text):
    # 剔除换行符和引号
    text = text.replace("\n", " ");
    text = text.replace("\"", "");

    # 保证每个标点符号都和前面的单词在一起
    # 这样不会被剔除,保留在马尔可夫链中
    punctuation = [',', '.', ';',':']
    for symbol in punctuation:
        text = text.replace(symbol, " "+symbol+" ");

    words = text.split(" ")
    # 过滤空单词
    words = [word for word in words if word != ""]

    wordDict = {}
    for i in range(1, len(words)):
        if words[i-1] not in wordDict:
                # 为单词新建一个词典
            wordDict[words[i-1]] = {}
        if words[i] not in wordDict[words[i-1]]:
            wordDict[words[i-1]][words[i]] = 0
        wordDict[words[i-1]][words[i]] = wordDict[words[i-1]][words[
                                                                i]] + 1

    return wordDict

text = str(urlopen("http://pythonscraping.com/files/inaugurationSpeech.txt")
```

```
            .read(), 'utf-8')
wordDict = buildWordDict(text)

# 生成链长为100的马尔可夫链
length = 100
chain = ""
currentWord = "I"
for i in range(0, length):
    chain += currentWord+" "
    currentWord = retrieveRandomWord(wordDict[currentWord])

print(chain)
```

代码的输出结果每次都会变化，下面是其中一个胡言乱语的结果：

```
I sincerely believe in Chief Magistrate to make all necessary sacrifices and
oppression of the remedies which we may have occurred to me in the arrangement
and disbursement of the democratic claims them , consolatory to have been best
political power in fervently commending every other addition of legislation , by
the interests which violate that the Government would compare our aboriginal
neighbors the people to its accomplishment . The latter also susceptible of the
Constitution not much mischief , disputes have left to betray . The maxim which
may sometimes be an impartial and to prevent the adoption or
```

那么代码是怎么实现的呢？

`buildWordDict` 函数把网上获取的演讲文本的字符串作为参数，然后对字符串做一些清理和格式化处理，去掉引号，把其他标点符号两端加上空格，这样就可以对每一个单词进行有效的处理。最后，再建立如下所示的一个二维字典——字典里有字典：

```
{word_a : {word_b : 2, word_c : 1, word_d : 1},
 word_e : {word_b : 5, word_d : 2},...}
```

在这个字典示例中，“word_a”出现了四次，有两次后面跟的单词是“word_b”，一次是“word_c”，一次是“word_d”。“word_e”出现了七次，有五次后面跟的单词是“word_b”，两次是“word_d”。

如果我们要画出这个结果的节点模型，那么“word_a”可能就有带 50% 概率的箭头指向“word_b”（四次中的两次），带 25% 概率的箭头指向“word_c”，还有带 25% 概率的箭头指向“word_d”。

一旦字典建成，不管你现在看到了文章的哪个词，都可以用这个字典作为查询表来选择下一个节点。[3] 这个字典的字典是这么使用的，如果我们现在位于“word_e”节点，那么下一

注 3：程序处理文本中的最后一个单词的下一个选择时可能会发生异常，因为这个单词后面没有单词。在我们的例子中，最后一个单词是点号（.），这样会很方便，因为它在演讲内容里一共出现了 215 次，所以选择下一个单词时不会出现问题。但是，在现实工作中，实现一个马尔可夫生成器时，文本的最后一个单词通常是需要慎重考虑的。

步就要把字典 {word_b : 5, word_d : 2} 传递到 retrieveRandomWord 函数。这个函数会按照字典中单词频次的权重随机获取一个单词。

先确定一个随机的开始词（示例中用的是经常使用的“I”），我们可以通过马尔可夫链随意地重复，生成我们需要的任意长度的句子。

维基百科六度分割：终结篇

在第 3 章，我们创建了收集从凯文 · 贝肯开始的维基词条链接的爬虫，最后存储在数据库里。为什么这里又把这个游戏搬出来？因为它体现了一种从一个页面指向另一个页面的链接路径选择问题（即找出 https://en.wikipedia.org/wiki/Kevin_Bacon 和 https://en.wikipedia.org/wiki/Eric_Idle 链接路径），这和从上面的马尔可夫链里找出一个单词到另一个单词的路径的问题是一样的。这类问题被称为*有向图*（directed graph）问题，其中 A → B 连通，并不意味着 B → A 同样连通。单词“football”后面可能经常跟的是单词“player”，但是单词“player”后面却很少跟着单词“football”。虽然凯文 · 贝肯（Kevin Bacon）的维基词条连接到他的老家费城（Philadelphia），但是费城的维基百科词条里却没有连接他的链接。

相反，原来的凯文 · 贝肯六度分割游戏是一个*无向图*（undirected graph）问题。例如凯文· 贝肯和朱莉娅 · 罗伯茨（Julia Roberts）共同演过电影《别闯阴阳界》（*Flatliners*，1990），因此凯文 · 贝肯词条通过《别闯阴阳界》的维基词条会链接到朱莉娅 · 罗伯茨词条，而朱莉娅 · 罗伯茨词条也会通过《别闯阴阳界》的维基词条链接到凯文 · 贝肯词条，两者的关系是相互的（就是没有“方向”性）。在计算机科学中，无向图问题相比有向图问题不太常见，两者都属于计算难题。

虽然解决这两类问题和对应的多个分支问题的方法有很多，但是在寻找有向图的最短路径问题中，即找出维基百科中凯文 · 贝肯词条和其他词条之间最短链接路径的方法中，效果最好且最常用的一种方法是*广度优先搜索*（breadth-first search）。

广度优先搜索算法的思路是优先搜寻直接连接到起始页的所有链接（而不是找到一个链接就纵向深入搜索）。如果这些链接不包含目标页面（你想要找的词条），就对第二层的链接——连接到起始页的页面的所有链接——进行搜索。这个过程不断重复，直到达到搜索深度限制（本例中使用的层数限制是 6 层）或者找到目标页面为止。

用第 5 章里获得的链接数据表，实现一个完整的广度优先搜索算法，代码如下所示：

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import pymysql

conn = pymysql.connect(host='127.0.0.1', unix_socket='/tmp/mysql.sock',
                       user='root', passwd=None, db='mysql', charset='utf8')
cur = conn.cursor()
```

```
cur.execute("USE wikipedia")

class SolutionFound(RuntimeError):
   def __init__(self, message):
      self.message = message

def getLinks(fromPageId):
    cur.execute("SELECT toPageId FROM links WHERE fromPageId = %s", (fromPageId))
    if cur.rowcount == 0:
        return None
    else:
        return [x[0] for x in cur.fetchall()]

def constructDict(currentPageId):
    links = getLinks(currentPageId)
    if links:
        return dict(zip(links, [{}]*len(links)))
    return {}

# 链接树要么为空,要么包含多个链接
def searchDepth(targetPageId, currentPageId, linkTree, depth):
    if depth == 0:
        # 停止递归,返回结果
        return linkTree
    if not linkTree:
        linkTree = constructDict(currentPageId)
        if not linkTree:
            # 若此节点页面无链接,则跳过此节点
            return {}
    if targetPageId in linkTree.keys():
        print("TARGET "+str(targetPageId)+" FOUND!")
        raise SolutionFound("PAGE: "+str(currentPageId))

    for branchKey, branchValue in linkTree.items():
        try:
            # 递归建立链接树
            linkTree[branchKey] = searchDepth(targetPageId, branchKey,
                                              branchValue, depth-1)
        except SolutionFound as e:
            print(e.message)
            raise SolutionFound("PAGE: "+str(currentPageId))
    return linkTree

try:
    searchDepth(134951, 1, {}, 4)
    print("No solution found")
except SolutionFound as e:
    print(e.message)
```

这里函数 getLinks 和 constructDict 是辅助函数，用来从数据库里获取给定页面的链接，然后把链接转换成字典。主函数 searchDepth 会递归地执行，同时构建和搜索链接树，一次搜索一层。其运行规则如下所示。

- 如果递归限制已经到达（即程序已经调用过很多次），就停止搜索，返回结果。
- 如果函数获取的链接字典是空的，就对当前页面的链接进行搜索。如果当前页面也没链接，就返回空链接字典。
- 如果当前页面包含我们搜索的页面链接，就把页面 ID 复制到递归的栈顶，然后抛出一个异常，显示页面已经找到。递归过程中的每个栈都会打印当前页面 ID，然后抛出异常显示页面已经找到，最终打印在屏幕上的就是一个完整的页面 ID 路径列表。
- 如果链接没找到，把递归限制减一，然后调用函数搜索下一层链接。

下面是凯文 · 贝肯词条（在数据库中页面 ID 为 1）和埃里克 · 艾德尔词条（在数据库中页面 ID 为 78520）的链接路径：

```
TARGET 134951 FOUND!
PAGE: 156224
PAGE: 155545
PAGE: 3
PAGE: 1
```

对应的链接名称是：Kevin Bacon → San Diego Comic Con International → Brian Froud → Terry Jones → Eric Idle。

除了解决“六度分隔”问题和对句子中的一个词后面跟哪个词问题进行建模，有向图和无向图还可以对网络数据采集中的许多场景进行建模。例如，一个网站与其他网站的链接关系是什么？一篇学术论文与其他学术论文之间的引用关系如何？零售网站上哪些产品应该捆绑销售？这个链接的强度是什么？这个链接是双向链接（reciprocal）吗？

了解这些基础知识对建模、可视化以及基于采集数据进行预测都非常有用。

8.3 自然语言工具包

到目前为止，本章主要讨论对文本中所有单词的统计分析。哪些单词使用得最频繁？哪些单词用得最少？一个单词后面跟着哪几个单词？这些单词是如何组合在一起的？我们应该做却还没做的事情，是理解每个单词的具体含义。

自然语言工具包（Natural Language Toolkit，NLTK）就是这样一个 Python 库，用于识别和标记英语文本中各个词的词性（parts of speech）。这个项目于 2000 年创建，经过 15 年的发展，由来自世界各地的几十个开发者共同努力维护。虽然它的功能非常丰富（有几本书专门介绍 NLTK），但这一节只重点介绍几个用法。

8.3.1 安装与设置

NLTK 模块的安装方法和其他 Python 模块一样，要么从 NLTK 网站直接下载安装包进行安装，要么用其他几个第三方安装器通过关键词“nltk”安装。详细的安装教程，请参考

NLTK 网站（http://www.nltk.org/install.html）。

模块安装之后，可以下载 NLTK 自带的文本库，这样你就可以非常轻松地实验 NLTK 的功能。在 Python 命令行输入下面的命令即可：

```
>>> import nltk
>>> nltk.download()
```

两行命令会打开 NLTK 的下载器（图 8-2）。

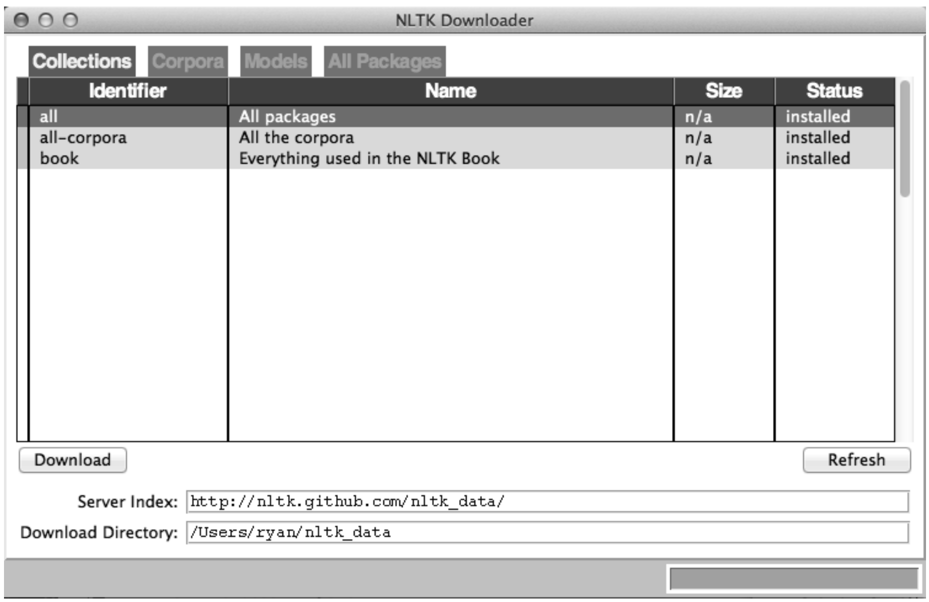


图 8-2：NLTK 下载器可以让你浏览和下载 NLTK 模块的包和文本库

推荐你安装所有的包。要下载的每个文件都是非常小的文本；你永远也不会知道后面会用到哪一个，而且任何时候都可以轻易地卸载它们。

8.3.2　用NLTK做统计分析

NLTK 很擅长生成一些统计信息，包括对一段文字的单词数量、单词频率和单词词性的统计。如果你只需要做一些简单直接的计算（比如，一段文字中不重复单词的数量），导入 NLTK 模块就太大材小用了——它是一个非常大的模块。但是，如果你还需要对文本做一些更有深度的分析，那么里面有许多函数可以帮你实现任何需要的统计指标。

用 NLTK 做统计分析一般是从 `Text` 对象开始的。`Text` 对象可以通过下面的方法用简单的 Python 字符串来创建：

```
from nltk import word_tokenize
from nltk import Text

tokens = word_tokenize("Here is some not very interesting text")
text = Text(tokens)
```

word_tokenize 函数的参数可以是任何 Python 字符串。如果你手边没有任何长字符串，但是还想尝试一些功能，在 NLTK 库里已经内置了几本书，可以用 import 函数导入：

```
from nltk.book import *
```

这样会加载九本书：

```
*** Introductory Examples for the NLTK Book ***
Loading text1, ..., text9 and sent1, ..., sent9
Type the name of the text or sentence to view it.
Type: 'texts()' or 'sents()' to list the materials.
text1: Moby Dick by Herman Melville 1851
text2: Sense and Sensibility by Jane Austen 1811
text3: The Book of Genesis
text4: Inaugural Address Corpus
text5: Chat Corpus
text6: Monty Python and the Holy Grail
text7: Wall Street Journal
text8: Personals Corpus
text9: The Man Who Was Thursday by G . K . Chesterton 1908
```

在后面的例子，我们都用 text6，“Monty Python and the Holy Grail”（一部 1975 年电影的剧本）。

文本对象可以像普通的 Python 数组那样操作，好像它们就是一个包含文本里所有单词的数组。用这个属性，你可以统计文本中不重复的单词，然后与总单词数据进行比较：

```
>>> len(text6)/len(words)
7.833333333333333
```

前面的数据表明剧本中每个单词平均被使用了八次。你还可以将文本对象放到一个频率分布对象 FreqDist 中，查看哪些单词是最常用的，以及单词的频率是多少。

```
>>> from nltk import FreqDist
>>> fdist = FreqDist(text6)
>>> fdist.most_common(10)
[(':', 1197), ('.', 816), ('!', 801), (',', 731), ("'", 421), ('[', 3
19), (']', 312), ('the', 299), ('I', 255), ('ARTHUR', 225)]
>>> fdist["Grail"]
34
```

因为这是一个剧本，所以剧本中创作的一些角色会显示出来。例如，全部大写的“ARTHUR”频繁地出现，因为它会出现在亚瑟王（King Arthur）每一句台词的前面。另外，分号（:）也出现在每一行的开头，当作分隔符把人物的姓名和人物的台词分开。根据这个特征，我们可以看到这个电影剧本一共有 1197 句台词！

在上一章我们已经用过的 2-gram 模型，在 NLTK 中称作 bigrams（你可能会听到有人把 3-gram 模型叫作“trigrams”，但是我个人还是更喜欢用 n-gram 而不是 bigram 或 trigram）。你可以用 NLTK 非常轻松地创建并搜索一个 2-gram 模型：

```
>>> from nltk import bigrams
>>> bigrams = bigrams(text6)
>>> bigramsDist = FreqDist(bigrams)
>>> bigramsDist[("Sir", "Robin")]
18
```

为了搜索 2-gram 序列“Sir Robin”，我们需要把它分解成一个数组 `("Sir", "Robin")`，用来匹配这个 2-gram 序列在频率分布中的表现方式。还有一个 `trigrams` 模块，它的工作方式完全相同。对于更一般的情形，你还可以导入 `ngrams` 模块：

```
>>> from nltk import ngrams
>>> fourgrams = ngrams(text6, 4)
>>> fourgramsDist = FreqDist(fourgrams)
>>> fourgramsDist[("father", "smelt", "of", "elderberries")]
1
```

这个 `ngrams` 函数被用来把文本对象分解成任意规模的 n-gram 序列，第二个参数决定规模大小。在这个例子中，我把文本分解成了 4-gram。然后，我就可以查出“father smelt of elderberries”这个短语在剧本中只出现了一次。

频率分布、文本对象和 n-gram 还可以整合在一个循环中进行迭代。例如，下面的程序就是打印文本中所有以“coconut”开头的 4-gram 序列：

```
from nltk.book import *
from nltk import ngrams
fourgrams = ngrams(text6, 4)
for fourgram in fourgrams:
    if fourgram[0] == "coconut":
        print(fourgram)
```

NLTK 库设计了许多不同的工具和对象来组织、统计、排序和度量大段文字的含义。尽管我们只是了解了 NLTK 函数用法的皮毛，但是大多数工具都设计得非常好，而且熟悉 Python 的人很容易操作它们。

8.3.3 用NLTK做词性分析

到现在为止，我们只是基于拼写方式对比和分类遇到的所有单词，并没有区分同义词或语境。

虽然有人可能会认为同义词不处理也基本没什么问题，但是如果你看到了它们使用的频率，可能会吓一跳。大多数以英语为母语的人可能不会注意到两个词互为同义词，也很少会考虑同一个词在不同的语境中可能会导致意思混乱。

"He was objective in achieving his objective of writing an objective philosophy, primarily using verbs in the objective case"（在实现他写作一本客观哲学书的目标时，他是客观的，因为他在描述客观情况时主要用动词）这句话容易被人类理解，但是网络爬虫可能会认为是同样的单词（objective）被用了四次，进而简单地忽略这四个单词各自不同的含义。

除了理清句子中各个词的词性，还要区分出某句话中一个单词的用法。例如，你可能会需要查找一些普通英文单词组成的公司名称，或者分析某个人对一个公司的评价，像"ACME Products is good"和"ACME Products is not bad"意思是一样的，即使一句话里用的是"good"，而另一句话用的是"bad"。

Penn Treebank 语义标记

NLTK 默认使用的语料标记系统是由美国宾夕法尼亚大学大学开发的颇受欢迎的 Penn Treebank 项目（http://www.cis.upenn.edu/~treebank/）中的语料标记部分。虽然有些标记意思明确（比如，CC 就是并列连接词，coordinating conjunction），有些标记却比较模糊（比如，RP 是一个小品词，particle）。下面是语义标记的对照表。

缩　写	全　称
CC	并列连接词（Coordinating conjunction）
CD	基数（Cardinal number）
DT	限定词（Determiner）
EX	表示存在性的"there"（Existential "there"）
FW	外来语（Foreign word）
IN	介词，从属连词（Preposition, subordinating conjunction）
JJ	形容词（Adjective）
JJR	形容词，比较级（Adjective, comparative）
JJS	形容词，最高级（Adjective, superlative）
LS	列表项标记符（List item marker）
MD	情态动词（Modal）
NN	名词，单数或不可数（Noun, singular or mass）
NNS	名词，复数（Noun, plural）
NNP	专有名词，单数（Proper noun, singular）
NNPS	专有名词，复数（Proper noun, plural）
PDT	前置限定词（Predeterminer）
POS	名词所有格 s 结尾（Possessive ending）
PRP	人称代词（Personal pronoun）
PRP$	物主代词（Possessive pronoun）
RB	副词（Adverb）
RBR	副词，比较级（Adverb, comparative）
RBS	副词，最高级（Adverb, superlative）

（续）

缩　写	全　称
RP	小品词（Particle）
SYM	符号（Symbol）
TO	介词“to”（“to”）
UH	感叹词（Interjection）
VB	动词，一般形式（Verb, base form）
VBD	动词，过去时（Verb, past tense）
VBG	动词，动名词或现在分词（Verb, gerund or present participle）
VBN	动词，过去分词（Verb, past participle）
VBP	动词，非第三人称单数（Verb, non-third person singular present）
VBZ	动词，第三人称单数（Verb, third person singular present）
WDT	Wh- 限定词（Wh-determiner）
WP	Wh- 代词（Wh-pronoun）
WP$	Wh- 物主代词（Possessive wh-pronoun）
WRB	Wh- 副词（Wh-adverb）

除了度量语言，NLTK 还可以用它的超级大字典分析文本内容，帮助人们寻找单词的含义。NLTK 的一个基本功能是识别句子中各个词的词性：

```
>>> from nltk.book import *
>>> from nltk import word_tokenize
>>> text = word_tokenize("Strange women lying in ponds distributing swords is no
basis for a system of government.  Supreme executive power derives from a mandate
from the masses, not from some farcical aquatic ceremony.")
>>> from nltk import pos_tag
>>> pos_tag(text)
[('Strange', 'NNP'), ('women', 'NNS'), ('lying', 'VBG'), ('in', 'IN')
, ('ponds', 'NNS'), ('distributing', 'VBG'), ('swords', 'NNS'), ('is'
, 'VBZ'), ('no', 'DT'), ('basis', 'NN'), ('for', 'IN'), ('a', 'DT'),
('system', 'NN'), ('of', 'IN'), ('government', 'NN'), ('.', '.'), ('S
upreme', 'NNP'), ('executive', 'NN'), ('power', 'NN'), ('derives', 'N
NS'), ('from', 'IN'), ('a', 'DT'), ('mandate', 'NN'), ('from', 'IN'),
 ('the', 'DT'), ('masses', 'NNS'), (',', ','), ('not', 'RB'), ('from'
, 'IN'), ('some', 'DT'), ('farcical', 'JJ'), ('aquatic', 'JJ'), ('cer
emony', 'NN'), ('.', '.')]
```

每个单词被分开放在一个元组中，一边是单词，一边是 NLTK 识别的词性标记（每个词性标记的具体含义请参考前面的 Penn Treebank 标记表）。虽然这看起来像是非常简单直接的查询，但是要正确地完成任务其实很复杂度，用下面的例子看更直观：

```
>>> text = word_tokenize("The dust was thick so he had to dust")
>>> pos_tag(text)
[('The', 'DT'), ('dust', 'NN'), ('was', 'VBD'), ('thick', 'JJ'), ('so
```

```
', 'RB'), ('he', 'PRP'), ('had', 'VBD'), ('to', 'TO'), ('dust', 'VB')
]
```

需要注意的是，“dust”在这句话里出现过两次：一次是名词，而另一次是动词。NLTK 可以基于句子的内容正确地识别出对应的用法。NLTK 用英语的上下文无关文法（context-free grammar）识别词性。上下文无关文法基本上可以看成一个规则集合，用一个有序的列表确定一个词后面可以跟哪些词。NLTK 的上下文无关文法确定的是一个词性后面可以跟哪些词性。无论什么时候，只要遇到像“dust”这样一个含义不明确的单词，NLTK 都会用上下文无关文法的规则来判断，然后确定一个合适的词性。

机器学习和机器训练

你也可以对 NLTK 进行训练，创建一个全新的上下文无关文法规则，比如，一种外语的上下文无关文法规则。如果你用 Penn Treebank 词性标记手工完成了那种语言的大部分文本的语义标记，那么你就可以把结果传给 NLTK，然后训练它对其他未标记的文本进行语义标记。在任何一个机器学习案例中，机器训练都是不可或缺的部分，这一点我们将在第 11 章训练爬虫识别验证码（CAPTCHA）时介绍。

那么，知道某段文字中一个词是动词还是名词有什么用呢？在计算机科学研究室里做研究可能非常好用，但是它对网络数据采集有什么用呢？

网络数据采集经常需要处理搜索的问题。你在采集了一个网站的文字之后，可能想从文字里面搜索“google”这个词，但你要的是作为动词的 google，不要作为专用名词的 Google。或者你就想查找 Google 公司的名称 Google，但是不想通过首字母大写来找出答案（人们可能忘记将首字母大写，直接写成 google）。那么这时函数 `pos_tag` 就很管用了：

```
from nltk import word_tokenize, sent_tokenize, pos_tag
sentences = sent_tokenize("Google is one of the best companies in the world.
I constantly google myself to see what I'm up to.")
nouns = ['NN', 'NNS', 'NNP', 'NNPS']

for sentence in sentences:
    if "google" in sentence.lower():
        taggedWords = pos_tag(word_tokenize(sentence))
            for word in taggleWords:
                if word[0].lower() == "google" and word[1] in nouns:
                    print(sentence)
```

这段代码只会打印包含单词“google”（或“Google”）作为名词而非动词的句子。当然，你也可以更明确地要求只打印标记是“NNP”（专用名词）的“Google”，但是 NLTK 有时也会判断错误，所以最好还是给自己留一些余地，具体情况由项目而定。

自然语言中的许多歧义问题都可以用 NLTK 的 `pos_tag` 解决。不只是搜索目标单词或短语，而是搜索带标记的目标单词或短语，这样可以大大提高爬虫搜索的准确率和效率。

8.4　其他资源

通过机器处理、分析和理解自然语言是计算机科学中最难的任务之一，在这个领域中有数不清的专著和学术论文。我希望这里介绍的一点儿内容可以让你的视野超越传统的网络数据采集，或至少在从事需要进行自然语言分析的项目时清楚应该从哪儿下手。

有许多非常优秀的学习资源专门介绍自然语言处理和 Python 的 NLTK。尤其是 Steven Bird、Ewan Klein 和 Edward Loper 合著的 *Natural Language Processing with Python*（http://shop.oreilly.com/product/9780596516499.do）对这个主题进行了详细的分析和介绍。

另外，James Pustejovsky 和 Amber Stubbs 合著的 *Natural Language Annotation for Machine Learning*（http://shop.oreilly.com/product/0636920020578.do），为自然语言处理提供了更深刻的理论指导。学习该书需要有 Python 基础，书中介绍的主题都可以用 Python 的 NLTK 完美地实现。

第 9 章

穿越网页表单与登录窗口进行采集

当你真正迈出网络数据采集基础之门的时候，遇到的第一个问题很可能是："我怎么获取登录窗口背后的信息呢？"今天，网络正在朝着页面交互、社交媒体、用户产生内容的趋势不断地演进。表单和登录窗口是许多网站中不可或缺的组成部分。不过，它们还是比较容易处理的。

到目前为止，我们示例中的网络爬虫在和大多数网站的服务器进行数据交互时，都是用 HTTP 协议的 `GET` 方法去请求信息。这一章，我们将重点介绍 `POST` 方法，即把信息推送给网络服务器进行存储和分析。

页面表单基本上可以看成是一种用户提交 `POST` 请求的方式，且这种请求方式是服务器能够理解和使用的。就像网站的 URL 链接可以帮助用户发送 `GET` 请求一样，HTML 表单可以帮助用户发出 `POST` 请求。当然，我们也可以用一点儿代码来自己创建这些请求，然后通过网络爬虫把它们提交给服务器。

9.1　Python Requests库

虽然用 Python 的标准库也可以控制网页表单，但是有时用一点儿语法糖可以让生活更甜蜜。当你想做比 `urllib` 库能够实现的基本 `GET` 请求更多的事情时，可以看看 Python 标准库之外的第三方库。

Requests 库（http://www.python-requests.org/）就是这样一个擅长处理那些复杂的 HTTP 请求、cookie、header（响应头和请求头）等内容的 Python 第三方库。

下面是 Requests 的创建者 Kenneth Reitz 对 Python 标准库工具的评价。

> Python 的标准库 `urllib2` 为你提供了大多数 HTTP 功能，但是它的 API 非常差劲。这是因为它是经过许多年一步步建立起来的——不同时期要面对的是不同的网络环境。于是为了完成最简单的任务，它需要耗费大量的工作（甚至要重写整个方法）。
>
> 事情不应该这样复杂，更不应该发生在 Python 里。

和任何 Python 第三方库一样，Requests 库也可以用其他第三方 Python 库管理器安装，比如 pip，或者直接下载源代码（https://github.com/kennethreitz/requests/tarball/master）安装。

9.2　提交一个基本表单

大多数网页表单都是由一些 HTML 字段、一个提交按钮、一个在表单处理完之后跳转的“执行结果”（表单属性 `action` 的值）页面构成。虽然这些 HTML 字段通常由文字内容构成，但是也可以实现文件上传或其他非文字内容。

因为大多数主流网站都会在它们的 robots.txt 文件里注明禁止爬虫接入登录表单（附录 C 介绍了采集这类表单的相关法律责任），所以为了安全起见，我在网站里放入了一组不同类型的表单和登录内容，这样你就可以用网络爬虫采集了。最简单的表单位于 http://pythonscraping.com/pages/files/form.html。

这个表单的源代码是：

```
<form method="post" action="processing.php">
First name: <input type="text" name="firstname"><br>
Last name: <input type="text" name="lastname"><br>
<input type="submit" value="Submit">
</form>
```

这里有几点需要注意一下：首先，两个输入字段的名称是 `firstname` 和 `lastname`，这一点非常重要。字段的名称决定了表单被确认后要被传送到服务器上的变量名称。如果你想模拟表单提交数据的行为，你就需要保证你的变量名称与字段名称是一一对应的。

还需要注意表单的真实行为其实发生在 processing.php（绝对路径是 http://pythonscraping.com/files/processing.php）。表单的任何 `POST` 请求其实都发生在这个页面上，并非表单本身所在的页面。切记：HTML 表单的目的，只是帮助网站的访问者发送格式合理的请求，向服务器请求没有出现的页面。除非你要对请求的设计样式进行研究，否则不需要花太多时间在表单所在的页面上。

用 Requests 库提交表单只用四行代码就可以实现，包括导入库文件和打印内容的语句（是的，就是这么简单）：

```
import requests

params = {'firstname': 'Ryan', 'lastname': 'Mitchell'}
r = requests.post("http://pythonscraping.com/files/processing.php", data=params)
print(r.text)
```

表单被提交之后，程序应该会返回执行页面的源代码，包括这行内容：

```
Hello there, Ryan Mitchell!
```

这个程序还可以用来处理许多网站的简单表单。比如 O’Reilly Media 新闻订阅页面的表单源代码如下所示：

```
<form action="http://post.oreilly.com/client/o/oreilly/forms/
              quicksignup.cgi" id="example_form2" method="POST">
    <input name="client_token" type="hidden" value="oreilly" />
    <input name="subscribe" type="hidden" value="optin" />
    <input name="success_url" type="hidden" value="http://oreilly.com/store/
                 newsletter-thankyou.html" />
    <input name="error_url" type="hidden" value="http://oreilly.com/store/
                 newsletter-signup-error.html" />
    <input name="topic_or_dod" type="hidden" value="1" />
    <input name="source" type="hidden" value="orm-home-t1-dotd" />
    <fieldset>
        <input class="email_address long" maxlength="200" name=
                     "email_addr" size="25" type="text" value=
                     "Enter your email here" />
        <button alt="Join" class="skinny" name="submit" onclick=
                       "return addClickTracking('orm','ebook','rightrail','dod'
                                                  );" value="submit">Join</button>
    </fieldset>
</form>
```

虽然第一次看这些会觉得恐怖，但是大多数情况下（后面我们会介绍异常）你只需要关注两件事：

- 你想提交数据的字段名称（在这个例子中是 email_addr）
- 表单的 action 属性，也就是表单提交后网站会显示的页面（在这个例子中是 http://post.oreilly.com/client/o/oreilly/forms/quicksignup.cgi）

把对应的信息增加到请求信息中，运行代码即可：

```
import requests
params = {'email_addr': 'ryan.e.mitchell@gmail.com'}
r = requests.post("http://post.oreilly.com/client/o/oreilly/forms/
                   quicksignup.cgi", data=params)
print(r.text)
```

在这个示例中，你真正加入 O’Reilly 的邮件列表之前，还要在网站上填写另一个表单，同样的代码也可以应用到需要填写的新表单上。不过，如果你自己在家做，希望你慎用这些

知识，不要给 O’Reilly 出版社提交无效的注册。

9.3　单选按钮、复选框和其他输入

显然，并非所有的网页表单都只是一堆文本字段和一个提交按钮。HTML 标准里提供了大量可用的表单字段：单选按钮、复选框和下拉选框等。在 HTML5 里面，还有其他的控件，像滚动条（范围输入字段）、邮箱、日期等。自定义的 JavaScript 字段可谓无所不能，可以实现取色器（colorpicker）、日历以及开发者能想到的任何功能。

无论表单的字段看起来多么复杂，仍然只有两件事是需要关注的：字段名称和值。字段名称可以通过查看源代码寻找 `name` 属性轻易获得。而字段的值有时会比较复杂，有可能是在表单提交之前通过 JavaScript 生成的。取色器就是一个比较奇怪的表单字段，它可能会用类似 `#F03030` 这样的值。

如果你不确定一个输入字段值的数据格式，有一些工具可以跟踪浏览器正在通过网站发出或接受的 `GET` 和 `POST` 请求的内容。之前提到过，跟踪 `GET` 请求效果最好也最直接的手段就是看网站的 URL 链接。如果 URL 链接像这样：

```
http://domainname.com?thing1=foo&thing2=bar
```

你明白这个请求就是下面这种表单：

```
<form method="GET" action="someProcessor.php">
<input type="someCrazyInputType" name="thing1" value="foo" />
<input type="anotherCrazyInputType" name="thing2" value="bar" />
<input type="submit" value="Submit" />
</form>
```

对应的 Python 参数就是：

```
{'thing1':'foo', 'thing2':'bar'}
```

具体内容如图 9-1 所示。

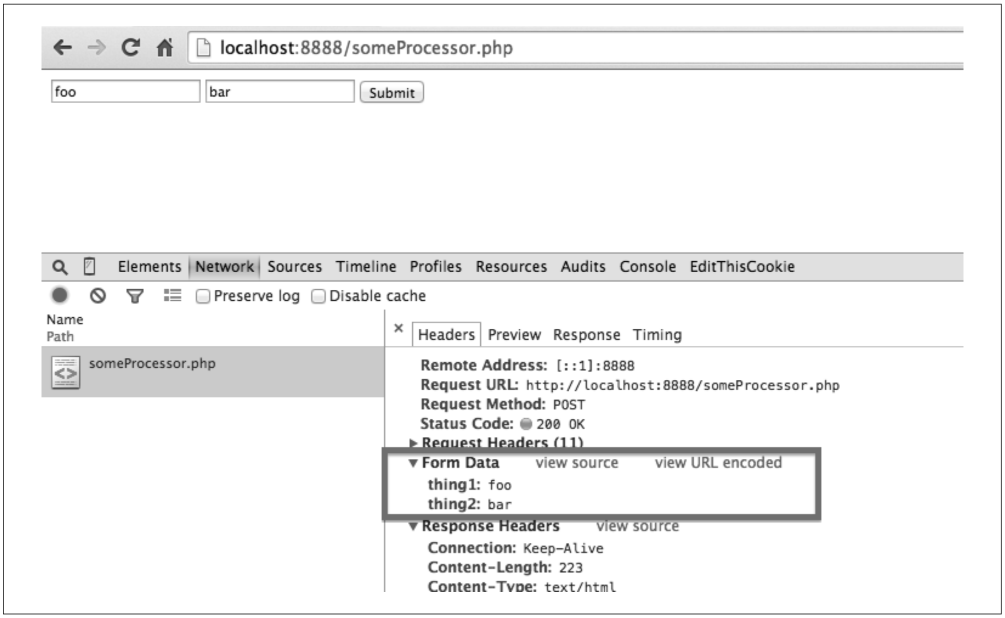


图 9-1：表单字段值如方框所示，其中 POST 请求参数 “thing1” 和 “thing2” 对应的值是 “foo” 和 “bar”

如果你遇到了一个看着比较复杂的 POST 表单，并且想查看浏览器向服务器传递了哪些参数，最简单的方法就是用 Chrome 浏览器的审查元素（inspector）或开发者工具查看。

Chrome 浏览器的开发者工具可以在菜单中通过更多工具→开发者工具打开（快捷键 F12）。它提供了浏览器与网站交互时产生的所有请求细节，是一种查看请求参数的好方法。

9.4 提交文件和图像

虽然上传文件在网络上很普遍，但是对于网络数据采集其实不太常用。但是，如果你想为自己网站的文件上传功能写一个测试实例，也是可以实现的。不管怎么说，掌握工作原理总是有用的。

在 http://pythonscraping/files/form2.html 有一个文件上传表单，页面上表单的源代码如下所示：

```
<form action="processing2.php" method="post" enctype="multipart/form-data">
  Submit a jpg, png, or gif: <input type="file" name="image"><br>
  <input type="submit" value="Upload File">
</form>
```

文件上传表单除了 <input> 标签里有一个 type 属性是 file，看起来和之前看到的文字字段的表单没什么两样。其实，Python Requests 库对这种表单的处理方式也和之前的非常相似：

```
import requests

files = {'uploadFile': open('../files/Python-logo.png', 'rb')}
r = requests.post("http://pythonscraping.com/pages/processing2.php",
                  files=files)
print(r.text)
```

需要注意，这里提交给表单字段 `uploadFile` 的值不是一个简单的字符串了，而是一个用 `open` 函数打开的 Python 文件对象。在这个例子中，我提交了一个保存在我电脑上的图像文件，文件路径是相对这个 Python 程序所在位置的 ../files/Python-logo.png。

没错，就是这么简单！

9.5 处理登录和cookie

到此为止，我们介绍过的大多数表单都允许你向网站提交信息，或者让你在提交表单后立即看到想要的页面信息。那么，这些表单和登录表单（当你浏览网站时让你保持“已登录”状态）有什么不同？

大多数新式的网站都用 cookie 跟踪用户是否已登录的状态信息。一旦网站验证了你的登录权证，它就会将它们保存在你的浏览器的 cookie 中，里面通常包含一个服务器生成的令牌、登录有效时限和状态跟踪信息。网站会把这个 cookie 当作信息验证的证据，在你浏览网站的每个页面时出示给服务器。在 20 世纪 90 年代中期广泛使用 cookie 之前，保证用户安全验证并跟踪用户是网站上的一大问题。

虽然 cookie 为网络开发者解决了大问题，但同时却为网络爬虫带来了大问题。你可以一整天只提交一次登录表单，但是如果你没有一直关注表单后来回传给你的那个 cookie，那么一段时间以后再次访问新页面时，你的登录状态就会丢失，需要重新登录。

我在 http://pythonscraping.com/pages/cookies/login.html 建了一个简单的登录表单（用户名可以是任意值，但是密码必须是“password”）。

这个表单在欢迎页面（http://pythonscraping.com/pages/cookies/welcome.php）处理，里面包含一个简介页面：http://pythonscraping.com/pages/cookies/profile.php。

如果在登录网站之前你想进入欢迎页面或者简介页面，会看到一个错误信息和访问前请先登录的指令。在简介页面中，网站会检测浏览器的 cookie，看它有没有页面已登录的设置信息。

用 Requests 库跟踪 cookie 同样很简单：

```
import requests

params = {'username': 'Ryan', 'password': 'password'}
```

```
r = requests.post("http://pythonscraping.com/pages/cookies/welcome.php", params)
print("Cookie is set to:")
print(r.cookies.get_dict())
print("-----------")
print("Going to profile page...")
r = requests.get("http://pythonscraping.com/pages/cookies/profile.php",
                 cookies=r.cookies)
print(r.text)
```

这里我向欢迎页面发送了一个登录参数，它的作用就像登录表单的处理器。然后我从请求结果中获取 cookie，打印登录状态的验证结果，然后再通过 `cookies` 参数把 cookie 发送到简介页面。

对简单的访问这样处理没有问题，但是如果你面对的网站比较复杂，它经常暗自调整 cookie，或者如果你从一开始就完全不想要用 cookie，该怎么处理呢？ Requests 库的 `session` 函数可以完美地解决这些问题：

```
import requests

session = requests.Session()

params = {'username': 'username', 'password': 'password'}
s = session.post("http://pythonscraping.com/pages/cookies/welcome.php", params)
print("Cookie is set to:")
print(s.cookies.get_dict())
print("-----------")
print("Going to profile page...")
s = session.get("http://pythonscraping.com/pages/cookies/profile.php")
print(s.text)
```

在这个例子中，会话（session）对象（调用 `requests.Session()` 获取）会持续跟踪会话信息，像 cookie、header，甚至包括运行 HTTP 协议的信息，比如 HTTPAdapter（为 HTTP 和 HTTPS 的链接会话提供统一接口）。

Requests 是一个非常给力的库，程序员完全不用费脑子，也不用写代码，可能只逊色于 Selenium（第 10 章将会介绍）。虽然写网络爬虫的时候，你可能想放手让 Requests 库替自己做所有的事情，但是持续关注 cookie 的状态，掌握它们可以控制的范围是非常重要的。这样可以避免痛苦地调试和追寻网站行为异常，节省很多时间。

HTTP基本接入认证

在发明 cookie 之前，处理网站登录最常用的方法就是用 HTTP 基本接入认证（HTTP basic access authentication）。有时还能见到它们，尤其是在一些安全性较高的网站或公司网站，以及一些 API 的使用上。我在 http://pythonscraping.com/pages/auth/login.php 用这种认证方法创建了一个页面（图 9-2）。

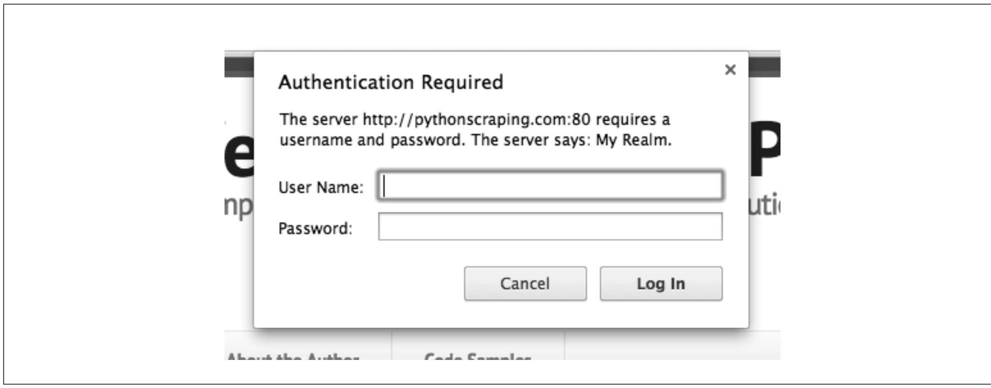


图 9-2：基本接入认证页面，用户必须通过用户名和密码才能登录

和前面的例子一样，你可以用任意用户名，但是密码必须是“password”。

Requests 库有一个 auth 模块专门用来处理 HTTP 认证：

```
import requests
from requests.auth import AuthBase
from requests.auth import HTTPBasicAuth

auth = HTTPBasicAuth('ryan', 'password')
r = requests.post(url="http://pythonscraping.com/pages/auth/login.php", auth=
                  auth)
print(r.text)
```

虽然这看着像是一个普通的 POST 请求，但是有一个 HTTPBasicAuth 对象作为 auth 参数传递到请求中。显示的结果将是用户名和密码验证成功的页面（如果验证失败，就是一个拒绝接入页面）。

9.6 其他表单问题

表单是网络恶意机器人（malicious bots）酷爱的网站切入点。你当然不希望机器人创建垃圾账号，占用昂贵的服务器资源，或者在博客上提交垃圾评论。因此，新式的网站经常在 HTML 中使用很多安全措施，让表单不能被快速穿越。

关于验证码（CAPTCHA）的作用，请查看第 11 章内容，里面介绍了 Python 的图像处理和文本识别方法。

如果你在提交表单的时候遇到了一个莫名其妙的错误，或者服务器一直以陌生的理由拒绝你，请查看第 12 章内容，里面介绍了蜜罐（honey pot）、隐含字段（hidden field），以及其他保护网页表单的安全措施。

第10章

采集JavaScript

客户端脚本语言是运行在浏览器而非服务器上的语言。客户端语言成功的前提是浏览器拥有正确地解释和执行这类语言的能力（这也是在浏览器上禁止 JavaScript 非常容易的原因）。

在一定程度上，由于很难让所有浏览器开发商都认可同一个标准，所以客户端语言比服务器端语言要少很多。不过这在网络数据采集的时候是件好事：要处理的语言越少越好。

通常，你在网上遇到的客户端语言只有两种：ActionScript（开发 Flash 应用的语言）和 JavaScript。今天 ActionScript 的使用率比 10 年前低很多，经常用于流媒体文件播放，用作在线游戏平台，或者是网站上那些没人想看更没人点击的“介绍”页面。总之，采集 Flash 页面的需要并不多，所以我重点介绍新式网页中普遍使用的客户端语言：JavaScript。

到目前为止，JavaScript 是网络上最常用也是支持者最多的客户端脚本语言。它可以收集用户的跟踪数据，不需要重载页面直接提交表单，在页面嵌入多媒体文件，甚至运行网页游戏。那些看起来非常简单的页面背后通常使用了许多 JavaScript 文件。你可以在网页源代码的 <script> 标签之间看到它们：

```
<script>
    alert("This creates a pop-up using JavaScript");
</script>
```

10.1 JavaScript简介

对要采集的语言预先做些了解会很有用。自己熟悉一下 JavaScript 总会有好处。

JavaScript 是一种弱类型语言，其语法通常可以与 C++ 和 Java 做对比。虽然语法中的一些元素，比如操作符、循环条件和数组，都与 C++、Java 语法很接近，但是 JavaScript 的弱类型和脚本形式被一些程序员看成是折磨人的怪兽。

例如，下面的 JavaScript 程序通过递归方式计算 Fibonacci 序列，最后把结果打印在浏览器的开发者控制台里：

```
<script>
function fibonacci(a, b){
    var nextNum = a + b;
    console.log(nextNum+" is in the Fibonacci sequence");
    if(nextNum < 100){
        fibonacci(b, nextNum);
    }
}
fibonacci(1, 1);
</script>
```

注意 JavaScript 里所有的变量都用 `var` 关键词字进行定义。这与 PHP 里的 `$` 符号类似，或者 Java 和 C++ 里的类型声明（`int`、`String`、`List` 等）。Python 不太一样，它没有这种显式的变量声明。

JavaScript 还有一个非常好的特性，就是把函数作为变量使用：

```
<script>
var fibonacci = function() {
    var a = 1;
    var b = 1;
    return function() {
        var temp = b;
        b = a + b;
        a = temp;
        return b;
    }
}
var fibInstance = fibonacci();
console.log(fibInstance()+" is in the Fibonacci sequence");
console.log(fibInstance()+" is in the Fibonacci sequence");
console.log(fibInstance()+" is in the Fibonacci sequence");
</script>
```

第一次看到这段代码可能有点儿头晕，不过如果你把这种特性看成 Lambda 表达式（第 2 章介绍过），就会很简单了。变量 `fibonacci` 被定义成一个函数。它的函数值返回一个递增的 Fibonacci 序列里较大的值。每次当它被调用时会返回 Fibonacci 的计算函数，再次执行序列计算，并增加函数变量的值。

虽然这样看起来有点儿复杂，但是在解决一些问题时，比如计算 Fibonacci 序列值，这种模式还是比较合适的。在处理用户行为和回调函数时，把函数作为变量进行传递是非常方

便的，另外在阅读 JavaScript 代码的时候必须适应这种编程方式。

常用JavaScript库

虽然了解 JavaScript 语言本身的语法很重要，但是在新式的网络中你必然要使用至少一种 JavaScript 语言的第三方库。在查看网页源代码的时候，你可能会看到很多常用的 JavaScript 库。

用 Python 执行 JavaScript 代码的效率非常低，既费时又费力，尤其是在处理规模较大的 JavaScript 代码时。如果有绕过 JavaScript 并直接解析它的方法（不需要执行它就可以获得信息）会非常实用，可以帮你避开一大堆 JavaScript 的麻烦事。

1. jQuery

jQuery 是一个十分常见的库，70% 最流行的网站（约 200 万）和约 30% 的其他网站（约 2 亿）都在使用。[1] 一个网站使用 jQuery 的特征，就是源代码里包含了 jQuery 入口，比如：

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"></
 script>
```

如果你在一个网站上看到了 jQuery，那么采集这个网站数据的时候要格外小心。jQuery 可以动态地创建 HTML 内容，只有在 JavaScript 代码执行之后才会显示。如果你用传统的方法采集页面内容，就只能获得 JavaScript 代码执行之前页面上的内容（我们将在 10.2 节详细介绍这个采集问题）。

另外，这些页面还可能包含动画、用户交互内容和嵌入式媒体，这些内容对网络数据采集都是挑战。

2. Google Analytics

有一半的网站都在用 Google Analytics[2]，它可能是网站最常用的 JavaScript 库和最受欢迎的用户跟踪工具。其实，http://pythonscraping.com 和 http://www.oreilly.com/ 都用了 Google Analytics。

很容易判断一个页面是不是使用了 Google Analytics。如果网站使用了它，在页面底部会有类似如下所示的 JavaScript 代码（取自 O’Reilly Media 网站）：

```
<!-- Google Analytics -->
<script type="text/javascript">
```

注 1：Dave Methvin 于 2014 年 1 月 13 日在他的博客中发表了“The State of jQuery 2014”（http://blog.jquery.com/2014/01/13/the-state-of-jquery-2014/）一文，里面包含了大量的统计数据。

注 2：W3Techs,“Usage Statistics and Market Share of Google Analytics for Websites”(<http://w3techs.com/technologies/details/ta-googleanalytics/all/all)。

```
var _gaq = _gaq || [];
_gaq.push(['_setAccount', 'UA-4591498-1']);
_gaq.push(['_setDomainName', 'oreilly.com']);
_gaq.push(['_addIgnoredRef', 'oreilly.com']);
_gaq.push(['_setSiteSpeedSampleRate', 50]);
_gaq.push(['_trackPageview']);

(function() { var ga = document.createElement('script'); ga.type =
'text/javascript'; ga.async = true; ga.src = ('https:' ==
document.location.protocol ? 'https://ssl' : 'http://www') +
'.google-analytics.com/ga.js'; var s =
document.getElementsByTagName('script')[0];
s.parentNode.insertBefore(ga, s); })();

</script>
```

如果一个网站使用了 Google Analytics 或其他类似的网络分析系统，而你不想让网站知道你在采集数据，就要确保把那些分析工具的 cookie 或者所有 cookie 都关掉。

3. Google地图

只要你上过网，就一定见过内嵌 Google 地图的网站。用 Google 地图的 API 很容易在任何网站上嵌入地图。

如果你要采集任何的位置数据，理解 Google 地图的工作方式可以让你轻松地获取格式规范的经纬度坐标和具体地址。在 Google 地图上，显示一个位置最常用的方法就是用标记（一个大头针）。

标记可以用下面的代码插在 Google 地图上：

```
var marker = new google.maps.Marker({
      position: new google.maps.LatLng(-25.363882,131.044922),
      map: map,
      title: 'Some marker text'
  });
```

Python 可以轻松地抽取出所有位置在 `google.maps.LatLng()` 里的坐标，生成一组经 / 纬度坐标值。

通过 Google 的“地理坐标反向查询”API（https://developers.google.com/maps/documentation/javascript/examples/geocoding-reverse），你可以把这些经纬度坐标组解析成格式规范的地址，便于存储和分析。

10.2　Ajax和动态HTML

到目前为止，我们与网站服务器通信的唯一方式，就是发出 HTTP 请求获取新页面。如果提交表单之后，或从服务器获取信息之后，网站的页面不需要重新刷新，那么你访问的网站就在用 Ajax 技术。

与一些人的印象不太一样，Ajax 其实并不是一门语言，而是用来完成网络任务（可以认为它与网络数据采集差不多）的一系列技术。Ajax 全称是 Asynchronous JavaScript and XML（异步 JavaScript 和 XML），网站不需要使用单独的页面请求就可以和网络服务器进行交互（收发信息）。需要注意的是：你不应该说“这个网站是 Ajax 写的”。正确的说法应该是“这个表单用 Ajax 与网络服务器通信”。

和 Ajax 一样，动态 HTML（dynamic HTML，DHTML）也是一系列用于解决网络问题的技术集合。DHTML 是用客户端语言改变页面的 HTML 元素（HTML、CSS，或者二者皆被改变）。比如，页面上的按钮只有当用户移动鼠标之后才出现，背景色可能每次点击都会改变，或者用一个 Ajax 请求触发页面加载一段新内容。

值得注意的是，虽然“动态”这个词往往和“移动”或“变化”联系在一起，但是那些使用了交互 HTML 组件、图像可以移动，或者带有嵌入式媒体文件的网页，并不一定就是动态 HTML，即使页面看起来是动态的。另外，一些表面看起来极其单调、静态的页面，底层却可能是用 DHTML 处理的，关键要看有没有用 JavaScript 控制 HTML 和 CSS 元素。

如果你采集过许多网站，很可能会遇到这样一种情况。你在浏览器上看到的内容，与你用爬虫从网站上采集的内容不一样。你可能会怀疑自己是不是哪个细节没处理好，希望找出内容采集不出来的原因。

有时你还会发现，网页用一个加载页面把你引到另一个页面上，但是网页的 URL 链接在这个过程中一直没有变化。

这些都是因为你的爬虫不能执行那些让页面产生各种神奇效果的 JavaScript 代码。如果网站的 HTML 页面没有运行 JavaScript，就可能和你在浏览器里看到的样子完全不同，因为浏览器可以正确地执行 JavaScript。

那些使用了 Ajax 或 DHTML 技术改变 / 加载内容的页面，可能有一些采集手段，但是用 Python 解决这个问题只有两种途径：直接从 JavaScript 代码里采集内容，或者用 Python 的第三方库运行 JavaScript，直接采集你在浏览器里看到的页面。

在Python中用Selenium执行JavaScript

Selenium（http://www.seleniumhq.org/）是一个强大的网络数据采集工具，其最初是为网站自动化测试而开发的。近几年，它还被广泛用于获取精确的网站快照，因为它们可以直接运行在浏览器上。Selenium 可以让浏览器自动加载页面，获取需要的数据，甚至页面截屏，或者判断网站上某些动作是否发生。

Selenium 自己不带浏览器，它需要与第三方浏览器结合在一起使用。例如，如果你在 Firefox 上运行 Selenium，可以直接看到一个 Firefox 窗口被打开，进入网站，然后执行你在代码中设置的动作。虽然这样可以看得更清楚，但是我更喜欢让程序在后台运行，所以

我用一个叫 PhantomJS（http://phantomjs.org/download.html）的工具代替真实的浏览器。

PhantomJS 是一个“无头”（headless）浏览器。它会把网站加载到内存并执行页面上的 JavaScript，但是它不会向用户展示网页的图形界面。把 Selenium 和 PhantomJS 结合在一起，就可以运行一个非常强大的网络爬虫了，可以处理 cookie、JavaScrip、header，以及任何你需要做的事情。

你可以从 PyPI 网站（https://pypi.python.org/simple/selenium/）下载 Selenium 库，也可以用第三方管理器（像 pip）用命令行安装。

PhantomJS 也可以从它的官方网站（http://phantomjs.org/download.html）下载。因为 PhantomJS 是一个功能完善（虽然无头）的浏览器，并非一个 Python 库，所以它不需要像 Python 的其他库一样安装，也不能用 pip 安装。

虽然有很多页面都用 Ajax 加载数据（尤其是 Google），我还是在 http://pythonscraping.com/pages/javascript/ajaxDemo.html 建了一个简单的页面来运行我们的爬虫。这个页面上有一些简单的文字，是手工敲在 HTML 代码里的，打开页面两秒钟之后，页面就会被替换成一个 Ajax 生成的内容。如果我们用传统的方法采集这个页面，只能获取加载前的页面，而我们真正需要的信息（Ajax 执行之后的页面）却抓不到。

Selenium 库是一个在 WebDriver 上调用的 API。WebDriver 有点儿像可以加载网站的浏览器，但是它也可以像 BeautifulSoup 对象一样用来查找页面元素，与页面上的元素进行交互（发送文本、点击等），以及执行其他动作来运行网络爬虫。

下面的代码可以获取前面测试页面上 Ajax“墙”后面的内容：

```
from selenium import webdriver
import time

driver = webdriver.PhantomJS(executable_path='')
driver.get("http://pythonscraping.com/pages/javascript/ajaxDemo.html")
time.sleep(3)
print(driver.find_element_by_id('content').text)
driver.close()
```

这段代码用 PhantomJS 库创建了一个新的 Selenium WebDriver，首先用 WebDriver 加载页面，然后暂停执行 3 秒钟，再查看页面获取（希望已经加载完成的）内容。

依据你的 PhantomJS 安装位置，在创建新的 PhantomJS WebDriver 的时候，你需要在 Selenium 的 WebDriver 接入点指明 PhantomJS 可执行文件的路径：

```
driver = webdriver.PhantomJS(executable_path='/path/to/download/
                             phantomjs-1.9.8-macosx/bin/phantomjs')
```

Selenium 的选择器

在之前的几章里，我们用过 BeautifulSoup 的选择器选择页面的元素，比如 find 和 findAll。Selenium 在 WebDriver 的 DOM 中使用了全新的选择器来查找元素，不过它们都使用非常直截了当的名称。

在这个例子中，我们用的选择器是 find_element_by_id，虽然下面的选择器也可以获取同样的结果：

```
driver.find_element_by_css_selector("#content")
driver.find_element_by_tag_name("div")
```

当然，如果你想选择页面上具有同样特征的多个元素，可以用 elements（换成复数）来返回一个 Python 列表：

```
driver.find_elements_by_css_selector("#content")
driver.find_elements_by_css_selector("div")
```

另外，如果你还是想用 BeautifulSoup 来解析网页内容，可以用 WebDriver 的 page_source 函数返回页面的源代码字符串。

```
pageSource = driver.page_source
bsObj = BeautifulSoup(pageSource)
print(bsObj.find(id="content").get_text())
```

如果程序配置都正确，上面的程序会在几秒钟以后显示下面的结果：

```
Here is some important text you want to retrieve!
A button to click!
```

需要注意的是，虽然页面里有一个元素是 HTML 按钮，但是 Selenium 的 .text 函数可以获取按钮的文本内容，就像它获取页面上其他元素内容的方式一样。

如果 time.sleep 的暂停时间由三秒改成一秒，那么上面程序采集的文本就会变成：

```
This is some content that will appear on the page while it's loading.
 You don't care about scraping this.
```

虽然这个方法奏效了，但是效率还不够高，在处理规模较大的网站时还是可能会出问题。页面的加载时间是不确定的，具体依赖于服务器某一毫秒的负载情况，以及不断变化的网速。虽然这个页面加载可能只需要花两秒多的时间，但是我们设置了三秒的等待时间以确保页面完全加载成功。一种更加有效的方法是，让 Selenium 不断地检查某个元素是否存在，以此确定页面是否已经完全加载，如果页面加载成功就执行后面的程序。

下面的程序用 id 是 loadedButton 的按钮检查页面是不是已经完全加载：

```
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
```

```
from selenium.webdriver.support import expected_conditions as EC

driver = webdriver.PhantomJS(executable_path='')
driver.get("http://pythonscraping.com/pages/javascript/ajaxDemo.html")
try:
    element = WebDriverWait(driver, 10).until(
                        EC.presence_of_element_located((By.ID, "loadedButton")))
finally:
    print(driver.find_element_by_id("content").text)
    driver.close()
```

程序里新导入了一些新的模块，最需要注意的就是 `WebDriverWait` 和 `expected_conditions`，这两个模块组合起来构成了 Selenium 的*隐式等待*（implicit wait）。

隐式等待与显式等待的不同之处在于，隐式等待是等 DOM 中某个状态发生后再继续运行代码（没有明确的等待时间，但是有最大等待时限，只要在时限内就可以），而显式等待明确设置了等待时间，如前面例子的等待三秒钟。在隐式等待中，DOM 触发的状态是用 `expected_conditions` 定义的（这里导入后用了别名 `EC`，是经常使用的简称）。在 Selenium 库里面元素被触发的期望条件（expected condition）有很多种，包括：

- 弹出一个提示框
- 一个元素被选中（比如文本框）
- 页面的标题改变了，或者某个文字显示在页面上或者某个元素里
- 一个元素在 DOM 中变成可见的，或者一个元素从 DOM 中消失了

当然，大多数的期望条件在使用前都需要你先指定等待的目标元素。元素用*定位器*（locator）指定。注意，定位器与选择器是不一样的（请看前面关于选择器的介绍）。定位器是一种抽象的查询语言，用 `By` 对象表示，可以用于不同的场合，包括创建选择器。

在下面的示例代码中，一个定位器被用来查找 id 是 `loadedButton` 的按钮：

```
EC.presence_of_element_located((By.ID, "loadedButton"))
```

定位器还可以用来创建选择器，配合 WebDriver 的 `find_element` 函数使用：

```
print(driver.find_element(By.ID, "content").text)
```

下面这行代码的功能和示例代码中一样：

```
print(driver.find_element_by_id("content").text)
```

如果你可以不用定位器，就不要用，毕竟这样可以少导入一个模块。但是，定位器是一种十分方便的工具，可以用在不同的应用中，并且具有很好的灵活性。

下面是定位器通过 `By` 对象进行选择的策略。

- `ID`

  在上面的例子里用过；通过 HTML 的 `id` 属性查找元素。

- `CLASS_NAME`

  通过 HTML 的 `class` 属性来查找元素。为什么这个函数是 `CLASS_NAME`，而不是简单的 `CLASS`？在 Selenium 的 Java 库里使用 `object.CLASS` 可能会出现问题，`.class` 是 Java 保留的一个方法。为了让 Selenium 语法可以兼容不同的语言，就用 `CLASS_NAME` 代替。

- `CSS_SELECTOR`

  通过 CSS 的 `class`、`id`、`tag` 属性名来查找元素，用 `#idName`、`.className`、`tagName` 表示。

- `LINK_TEXT`

  通过链接文字查找 HTML 的 `<a>` 标签。例如，如果一个链接的文字是“Next”，就可以用 `(By.LINK_TEXT, "Next")` 来选择。

- `PARTIAL_LINK_TEXT`

  与 `LINK_TEXT` 类似，只是通过部分链接文字来查找。

- `NAME`

  通过 HTML 标签的 `name` 属性查找。这在处理 HTML 表单时非常方便。

- `TAG_NAME`

  通过 HTML 标签的名称查找。

- `XPATH`

  用 `XPath` 表达式（语法在下面介绍）选择匹配的元素。

XPath 语法

XPath（XML Path，XML 路径）是在 XML 文档中导航和选择元素的查询语言。它由 W3C 于 1999 年创建，在 Python、Java 和 C# 这些语言中有时会用 XPath 来处理 XML 文档。

虽然 BeautifulSoup 不支持 XPath，但是本书中的很多库（lxml、Selenium、Scrapy 等）都支持。它的使用方式通常和 CSS 选择器（比如 `mytag#idname`）一样，虽然它原本被设计用于处理更规范的 XML 文档而不是 HTML 文档。

在 XPath 语法中有四个重要概念。

- 根节点和非根节点
  - `/div` 选择 div 节点，只有当它是文档的根节点时
  - `//div` 选择文档中所有的 div 节点（包括非根节点）

- 通过属性选择节点
  - //@href 选择带 href 属性的所有节点
  - //a[@href='http://google.com'] 选择页面中所有指向 Google 网站的链接
- 通过位置选择节点
  - //a[3] 选择文档中的第三个链接
  - //table[last()] 选择文档中的最后一个表
  - //a[position() < 3] 选择文档中的前三个链接
- 星号（*）匹配任意字符或节点，可以在不同条件下使用
  - //table/tr/* 选择所有表格行 tr 标签的所有的子节点（这很适合选择 th 和 td 标签）
  - //div[@*] 选择带任意属性的所有 div 标签

当然，XPath 还有很多高级的语法特征。经过这些年的发展，它已经变成一种非常复杂的查询语言，可以使用布尔类型、函数（如 position()），以及大量这里没介绍的操作符。

如果这里介绍的几个 XPath 功能解决不了你的 HTML 或 XML 元素选择问题，请参考微软的 XPath 语法页面（https://msdn.microsoft.com/en-us/enus/library/ms256471）。

10.3 处理重定向

客户端重定向是在服务器将页面内容发送到浏览器之前，由浏览器执行 JavaScript 完成的页面跳转，而不是服务器完成的跳转。当使用浏览器访问页面的时候，有时很难区分这两种重定向。由于客户端重定向执行很快，加载页面时你甚至感觉不到任何延迟，所以会让你觉得这个重定向就是一个服务器端重定向。

但是，在进行网络数据采集的时候，这两种重定向的差异是非常明显的。根据具体情况，服务器端重定向一般都可以轻松地通过 Python 的 urllib 库解决，不需要使用 Selenium（更多的介绍请参考第 3 章）。客户端重定向却不能这样处理，除非你有工具可以执行 JavaScript。

Selenium 可以执行这种 JavaScript 重定向，和它处理其他 JavaScript 的方式一样；但是这类重定向的主要问题是什么时候停止页面监控，也就是说，怎么识别一个页面已经完成重定向。在 http://pythonscraping.com/pages/javascript/redirectDemo2.html 的示例页面是客户端重定向的例子，有两秒的延迟。

我们可以用一种智能的方法来检测客户端重定向是否完成，首先从页面开始加载时就“监视”DOM 中的一个元素，然后重复调用这个元素直到 Selenium 抛出一个 StaleElementReferenceException 异常；也就是说，元素不在页面的 DOM 里了，说明这时网站已经跳转：

```python
from selenium import webdriver
import time
from selenium.webdriver.remote.webelement import WebElement
from selenium.common.exceptions import StaleElementReferenceException

def waitForLoad(driver):
    elem = driver.find_element_by_tag_name("html")
    count = 0
    while True:
        count += 1
        if count > 20:
            print("Timing out after 10 seconds and returning")
            return
        time.sleep(.5)
        try:
            elem == driver.find_element_by_tag_name("html")
        except StaleElementReferenceException:
            return

driver = webdriver.PhantomJS(executable_path='<Path to Phantom Js>')
driver.get("http://pythonscraping.com/pages/javascript/redirectDemo1.html")
waitForLoad(driver)
print(driver.page_source)
```

这个程序每半分钟检查一次网页，看看 `html` 标签还在不在，时限为 10 秒钟，不过检查的时间间隔和时限都可以根据实际情况随意调整。

第 11 章

图像识别与文字处理

从 Google 的无人驾驶汽车到可以识别假钞的自动售卖机，机器视觉一直都是一个应用广泛且具有深远的影响和雄伟的愿景的领域。在这一章里，我们将重点介绍机器视觉的一个分支：文字识别，介绍如何用一些 Python 库来识别和使用在线图片中的文字。

当你不想让自己的文字被网络机器人采集时，把文字做成图片放在网页上是常用的办法。在一些联系人通讯录里经常可以看到，一个邮箱地址被部分或全部转换成图片。人们可能觉察不出明显的差异，但是机器人阅读这些图片会非常困难，这种方法可以防止多数垃圾邮件发送器轻易地获取你的邮箱地址。

利用这种人类用户可以正常读取但是大多数机器人都没法读取的图片，验证码（CAPTCHA）就出现了。验证码读取的难易程度也大不相同，有些验证码比其他的更加难读，后面我们会介绍这种问题。

但是，验证码并不是网络爬虫数据采集时需要进行图像转文字翻译工作的唯一对象。目前，有很多文档都是简单地扫描后直接放到网上，它们和互联网上的很多文档一样都没法儿直接使用，尽管它们都“近在眼前”。如果无法将图像转为文字，要想使用这些文档的内容，就只能人工手敲了——没人愿意花时间干这事儿。

将图像翻译成文字一般被称为*光学文字识别*（Optical Character Recognition，OCR）。可以实现 OCR 的底层库并不多，目前很多库都是使用共同的几个底层 OCR 库，或者是在上面进行定制。这类 OCR 系统有时会变得非常复杂，所有我建议你在实践这一章的代码示例之前先阅读下一节的内容。

11.1 OCR库概述

在读取和处理图像、图像相关的机器学习以及创建图像等任务中，Python 一直都是非常出色的语言。虽然有很多库可以进行图像处理，但在这里我们只重点介绍两个库：Pillow 和 Tesseract。

每个库都可以从它们的网站上下载并安装（http://pillow.readthedocs.org/installation.html 和 https://pypi.python.org/pypi/pytesseract），或者用第三方管理器（像 pip）通过“pillow”和“pytesseract”进行安装。

11.1.1 Pillow

尽管 Pillow 算不上是图像处理功能最全的库，但是它拥有你需要使用的全部功能，除非你要用 Python 重写一个 Photoshop 或进行更加复杂的研究。它也是一个文档健全且十分易用的库。

Pillow 是从 Python 2.x 版本的 Python 图像库（Python Imaging Library，PIL）分出来的，支持 Python 3.x 版本。和 PIL 一样，Pillow 也可以轻松地导入代码，并通过大量的过滤、修饰甚至像素级的变换操作处理图片：

```
from PIL import Image, ImageFilter

kitten = Image.open("kitten.jpg")
blurryKitten = kitten.filter(ImageFilter.GaussianBlur)
blurryKitten.save("kitten_blurred.jpg")
blurryKitten.show()
```

在上面这个例子中，图片 kitten.jpg 会在默认的图片浏览器里打开，不过看着会有点儿模糊。之后这个比较模糊的图片被另存为 kitten_blurred.jpg，与原图放在一个文件夹里。

我们可以用 Pillow 完成图片的预处理，让机器可以更方便地读取图片。除了这些简单的事情之外，Pillow 还可以完成许多复杂的图像处理工作。更多的信息，请查看 Pillow 文档（http://pillow.readthedocs.org/）。

11.1.2 Tesseract

Tesseract 是一个 OCR 库，目前由 Google 赞助（Google 也是一家以 OCR 和机器学习技术闻名于世的公司）。Tesseract 是目前公认最优秀、最精确的开源 OCR 系统。

除了极高的精确度，Tesseract 也具有很高的灵活性。它可以通过训练识别出任何字体（只要这些字体的风格保持不变就可以，后面我们会介绍），也可以识别出任何 Unicode 字符。

和本书前面提到的那些库不同，Tesseract 是一个 Python 的命令行工具，不是通过 `import`

语句导入的库。安装之后，要用 `tesseract` 命令在 Python 的外面运行。

安装Tesseract

在 Windows 系统上，下载方便的可执行安装文件（https://code.google.com/p/tesseract-ocr/downloads/list）安装即可。写到这里的时候，最新的版本是 3.02，新版本应该也可以这样安装。

Linux 用户可以通过 `apt-get` 安装：

```
$sudo apt-get tesseract-ocr
```

在 Mac 上安装 Tesseract 有点儿复杂，不过用 Homebrew（http://brew.sh/）等第三方库可以很方便地安装。Homebrew 在第 5 章介绍 MySQL 安装过程时提到过。例如，你可以用下面两行代码首先安装 Homebrew，然后再安装 Tesseract：

```
$ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/ \
          install/master/install)"
$brew install tesseract
```

也可以从 Tesseract 项目下载页面（https://code.google.com/p/tesseract-ocr/downloads/list）下载源代码安装。

要使用 Tesseract 的功能，比如后面的示例中训练程序识别字母，你需要先在系统中设置一个新的环境变量 `$TESSDATA_PREFIX`，让 Tesseract 知道训练的数据文件存储在哪里。

在大多数 Linux 系统和 Mac OS X 系统上，你可以这么设置：

```
$export TESSDATA_PREFIX=/usr/local/share/
```

值得注意的是，虽然 `/usr/local/share/` 是 Tesseract 的默认数据存储位置，但是你还是应该仔细地检查一下，确保自己的安装没问题。

在 Windows 系统上也类似，你可以通过下面这行命令设置环境变量：

```
#setx TESSDATA_PREFIX C:\Program Files\Tesseract OCR\
```

11.1.3 NumPy

虽然 NumPy 并非解决 OCR 问题时必须使用的库，但是如果你想训练 Tesseract 识别本章后面提到的字符或字体，那么就会用到它。NumPy 是一个非常强大的库，具有大量线性代数以及大规模科学计算的方法。因为 NumPy 可以用数学方法把图片表示成巨大的像素数组，所以它可以流畅地配合 Tesseract 完成任务。

和其他 Python 库一样，NumPy 可以通过第三方包管理器（比如 pip）来安装：

```
$pip install numpy
```

11.2 处理格式规范的文字

你要处理的大多数文字都是比较干净、格式规范的。格式规范的文字通常可以满足一些需求，不过究竟什么是“格式混乱”，什么算“格式规范”，确实因人而异。

通常，格式规范的文字具有以下特点：

- 使用一个标准字体（不包含手写体、草书，或者十分“花哨的”字体）
- 虽然被复印或拍照，字体还是很清晰，没有多余的痕迹或污点
- 排列整齐，没有歪歪斜斜的字
- 没有超出图片范围，也没有残缺不全，或紧紧贴在图片的边缘

文字的一些格式问题在图片预处理时可以进行解决。例如，可以把图片转换成灰度图，调整亮度和对比度，还可以根据需要进行裁剪和旋转。但是，这些做法在进行更具扩展性的训练时会遇到一些限制。详情请见 11.3 节。

图 11-1 是格式规范文字的理想示例。

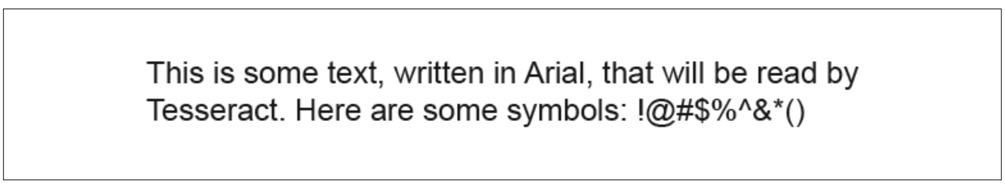


图 11-1：样本文字被保存为 tif 文件，将由 Tesseract 读取

你可以通过下面的命令运行 Tesseract，读取文件并把结果写到一个文本文件中：

```
$tesseract text.tif textoutput | cat textoutput.txt
```

输出结果的第一行是 Tesseract 的版本信息，表明它正在运行，后面是图片识别结果 textoutput.txt 文件里的内容：

```
Tesseract Open Source OCR Engine v3.02.02 with Leptonica
This is some text, written in Arial, that will be read by
Tesseract. Here are some symbols: !@#$%"&'()
```

你会发现识别结果很准确，不过符号“^”和“*”分别被表示成了双引号和单引号。大体上可以让你很舒服地阅读。

图片先进行模糊处理，转换成一个 JPG 压缩格式的图片，再增加一点儿背景渐变，识别效果就会变得很差（如图 11-2 所示）。

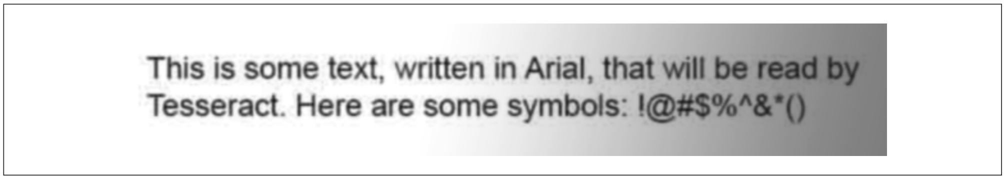


图 11-2：你在网上看到的许多图片可能都像这样

Tesseract 不能完整处理这个图片，主要是因为图片背景色是渐变的，最终结果是这样：

```
This is some text, written In Arlal, that"
Tesseract. Here are some symbols: _
```

你会发现，随着背景色从左到右不断加深，文字变得越来越难以识别，Tesseract 识别出的每一行的最后几个字符都是错的。另外，经过 JPG 格式转换和模糊效果处理，Tesseract 更难识别小写“i”和大写“I”以及数字“1”。

遇到这类问题，可以先用 Python 脚本对图片进行清理。利用 Pillow 库，我们可以创建一个阈值过滤器来去掉渐变的背景色，只把文字留下来，从而让图片更加清晰，便于 Tesseract 读取：

```
from PIL import Image
import subprocess

def cleanFile(filePath, newFilePath):
    image = Image.open(filePath)

    # 对图片进行阈值过滤,然后保存
    image = image.point(lambda x: 0 if x<143 else 255)
    image.save(newFilePath)

    # 调用系统的tesseract命令对图片进行OCR识别
    subprocess.call(["tesseract", newFilePath, "output"])

    # 打开文件读取结果
    outputFile = open("output.txt", 'r')
    print(outputFile.read())
    outputFile.close()

cleanFile("text_2.jpg", "text_2_clean.png")
```

自动创建的 text_2_clean.png，如下图所示。

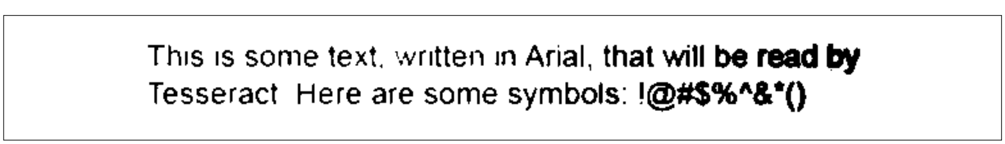


图 11-3：通过一个阈值对前面的“模糊”图片进行过滤的结果

除了一些标点符号不太清晰或丢失了，大部分文字都被读出来了。Tesseract 给出了最好的结果：

```
This us some text' written In Anal, that will be read by
Tesseract Here are some symbols: !@#$%"&'()
```

图片上的点和逗号经过处理后变得非常小，不论从我们的视角还是 Tesseract 的视角看，这些都从图片上基本消失了。还有一点失误是把“Arial”看成了“Anal”，这是 Tesseract 把“r”和“i”都解释成了“n”的结果。

不过，相比上一版本被截断的识别结果，这版算有很大进步了。

Tesseract 最大的缺点是对渐变背景色的处理。之前那个版本中，Tesseract 的算法在读取文字之前自动尝试调整图片对比度，但是如果你用 Pillow 库这样的工具对图片进行预处理，效果会更好。

在提交给 Tesseract 处理之前，那些带标题的、带有大片空白的图片，或者有其他问题的图片，都应该做预处理。

从网站图片中抓取文字

用 Tesseract 读取硬盘里图片上的文字，可能不怎么令人兴奋，但当我们把它和网络爬虫组合使用时，就能成为一个强大的工具。网站上的图片可能并不是故意把文字做得很花哨（就像餐馆菜单的 JPG 图片上的艺术字），但它们上面的文字对网络爬虫来说就是隐藏起来了，我将在下一个例子里演示。

虽然亚马逊的 robots.txt 文件允许抓取网站的产品页面，但是图书的预览页通常不让网络机器人采集。图书的预览页是通过用户触发 Ajax 脚本进行加载的，预览图片隐藏在 div 节点下面；其实，普通的访问者会觉得它们看起来更像是一个 Flash 动画，而不是一个图片文件。当然，即使我们能获得图片，要把它们读成文字也没那么简单。

下面的程序就解决了这个问题：首先导航到托尔斯泰的《战争与和平》的大字号印刷版[1]，打开阅读器，收集图片的 URL 链接，然后下载图片，识别图片，最后打印每个图片的文字。因为这个程序很复杂，利用了前面几章的多个程序片段，所以我增加了一些注释以让每段代码的目的更加清晰：

```python
import time
from urllib.request import urlretrieve
import subprocess
```

注 1：当处理那些没有训练过的文字时，Tesseract 对大字号印刷版书籍版本的识别效果更好，尤其是在图片比较小的时候。在下一节我们将介绍如何用不同的字体训练 Tesseract，这样可以帮它识别更小的字，包括普通字号印刷版书籍。

```
from selenium import webdriver

# 创建新的Selenium driver
driver = webdriver.PhantomJS(executable_path='<Path to Phantom JS>')
# 有时我发现PhantomJS查找元素有问题，但是Firefox没有。
# 如果你运行程序的时候出现问题，去掉下面这行注释，
# 用Selenium试试Firefox浏览器：
# driver = webdriver.Firefox()

driver.get(
      "http://www.amazon.com/War-Peace-Leo-Nikolayevich-Tolstoy/dp/1427030200")
time.sleep(2)

# 单击图书预览按钮
driver.find_element_by_id("sitbLogoImg").click()
imageList = set()

# 等待页面加载完成
time.sleep(5)
# 当向右箭头可以点击时，开始翻页
while "pointer" in driver.find_element_by_id("sitbReaderRightPageTurner")
                                            .get_attribute("style"):
    driver.find_element_by_id("sitbReaderRightPageTurner").click()
    time.sleep(2)
    # 获取已加载的新页面（一次可以加载多个页面，但是重复的页面不能加载到集合中）
    pages = driver.find_elements_by_xpath("//div[@class='pageImage']/div/img")
    for page in pages:
        image = page.get_attribute("src")
        imageList.add(image)

driver.quit()

# 用Tesseract处理我们收集的图片URL链接
for image in sorted(imageList):
    urlretrieve(image, "page.jpg")
    p = subprocess.Popen(["tesseract", "page.jpg", "page"],
                         stdout=subprocess.PIPE,stderr=subprocess.PIPE)
    p.wait()
    f = open("page.txt", "r")
    print(f.read())
```

和我们前面使用 Tesseract 读取的效果一样，这个程序也会完美地打印书中很多长长的段落，第六页的预览如下所示：

```
6

"A word of friendly advice, mon
cher. Be off as soon as you can,
that's all I have to tell you. Happy
he who has ears to hear. Good-by,
my dear fellow. Oh, by the by!" he
shouted through the doorway after
Pierre, "is it true that the countess
has fallen into the clutches of the
```

```
holy fathers of the Society of je-
sus?"

Pierre did not answer and left Ros-
topchin's room more sullen and an-
gry than he had ever before shown
himself.
```

但是，当文字出现在彩色封面上时，结果就不那么完美了：

```
WEI' nrrd Peace
Len Nlkelayevldu Iolfluy

Readmg shmdd be ax
wlnvame asnossxble Wenfler
an mm m our cram: Llhvary

— Leo Tmsloy was a Russian rwovelwst
I and moval phflmopher med lur
A ms Ideas 01 nonviolenx reswslance m 5 We range 0, "and"
```

当然，你可以用 Pillow 库挑选图片进行清理，但是如果想把文字加工成普通人可以看懂的效果，还需要花很多时间去处理。

下一节我们将介绍另一种方法来解决文字混乱的问题，尤其是当你愿意花一点儿时间训练 Tesseract 的时候。通过给 Tesseract 提供大量已知的文字与图片映射集，经过训练 Tesseract 就可以"学会"识别同一种字体，而且可以达到极高的精确率和准确率，甚至可以忽略图片中文字的背景色和相对位置等问题。

11.3 读取验证码与训练Tesseract

虽然大多数人对单词"CAPTCHA"都很熟悉，但是很少人知道它的具体含义：全自动区分计算机和人类的图灵测试（Completely Automated Public Turing test to tell Computers and Humans Apart）。它的奇怪缩写似乎表示，它一直在扮演着十分奇怪的角色。其目的是为了阻止网站访问，而不是让访问更通畅，它经常让人类和非人类的网络机器人深陷验证码识别的泥潭不能自拔。

图灵测试首次出现在阿兰·图灵（Alan Turing）1950 年发表的论文"计算装置与智能"（Computing Machinery and Intelligence）中。他在论文中描述了这样一种场景：一个人可以和其他人交流，也可以通过计算机终端和人工智能程序交流。如果一番对话之后这个人不能区分人和人工智能程序，那么就认为这个人工智能程序通过了图灵测试，图灵认为这个人工智能程序就可以真正地"思考"所有的事情。

令人啼笑皆非的是，60 多年以后，我们开始用这些原本测试程序的题目来测试我们自己。Google 的 reCAPTCHA 难得令人发指，作为目前最具有安全意识的流行网站，Google 拦截

了多达 25% 的准备访问网站的正常人类用户。[2]

大多数其他的验证码都是比较简单的。例如，流行的 PHP 内容管理系统 Drupal 有一个著名的验证码模块（https://www.drupal.org/project/captcha），可以生成不同难度的验证码。默认图片如图 11-4 所示。

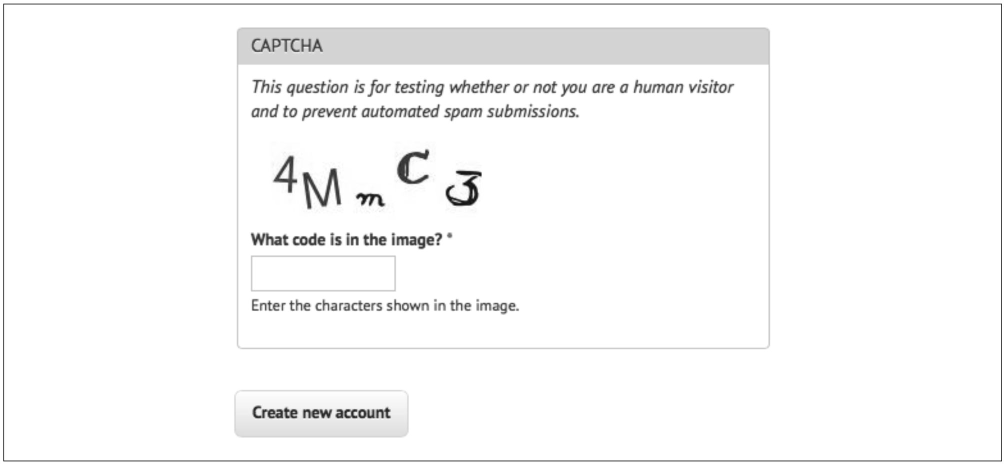


图 11-4：Drupal 的验证码项目的默认文字验证码示例

那么与其他验证码相比，究竟是什么让这个验证码更容易被人类和机器读懂呢？

- 字母没有相互叠加在一起，在水平方向上也没有彼此交叉。也就是说，可以在每一个字母外面画一个方框，而不会重叠在一起。
- 图片没有背景色、线条或其他对 OCR 程序产生干扰的噪点。
- 虽然不能因一个图片下定论，但是这个验证码用的字体种类很少，而且用的是 sans-serif 字体（像“4”和“M”）和一种手写形式的字体（像“m”“C”和“3”）。
- 白色背景色与深色字母之间的对比度很高。

这个验证码只做了一点点改变，就让 OCR 程序很难识别。

- 字母和数据都使用了，这会增加待搜索字符的数量。
- 字母随机的倾斜程度会迷惑 OCR 软件，但是人类还是很容易识别的。
- 那个比较陌生的手写字体很有挑战性，在“C”和“3”里面还有额外的线条。另外这个非常小的小写“m”，计算机需要进行额外的训练才能识别。

用下面的代码运行 Tesseract 识别图片：

注 2：详情请见 Quora 问题：https://www.quora.com/What-is-the-success-rate-of-legitimate-users-passing-reCAPTCHA-tests。

```
$tesseract captchaExample.png output
```

我们得到的结果 output.txt 是：

```
4N\,,,C<3
```

虽然识别出了 4、C 和 3，但是显然这样的结果永远也不能识别出正确的验证码。

训练Tesseract

要训练 Tesseract 识别一种文字，无论是晦涩难懂的字体还是验证码，你都需要向 Tesseract 提供每个字符不同形式的样本。

做这个枯燥的工作可能要花好几个小时的时间，你可能更想用这点儿时间找个好看的视频或电影看看。首先要把大量的验证码样本下载到一个文件夹里。下载的样本数量由验证码的复杂程度决定；我在训练集里一共放了 100 个样本（一共 500 个字符，平均每个字符 8 个样本；a~z 大小写字母加 0~9 数字，一共 62 个字符），应该足够训练的了。

提示：建议使用验证码的真实结果给每个样本文件命名（即 4MmC3.jpg）。这样可以帮你一次性对大量的文件进行快速检查——你可以先把图片调成缩略图模式，然后通过文件名对比不同的图片。这样在后面的步骤中进行训练效果的检查也会很方便。

第二步是准确地告诉 Tesseract 一张图片中的每个字符是什么，以及每个字符的具体位置。这里需要创建一些矩形定位文件（box file），一个验证码图片生成一个矩形定位文件。一个图片的矩形定位文件如下所示：

```
4 15 26 33 55 0
M 38 13 67 45 0
m 79 15 101 26 0
C 111 33 136 60 0
3 147 17 176 45 0
```

第一列符号是图片中的每个字符，后面的 4 个数字分别是包围这个字符的最小矩形的坐标（图片左下角是原点 (0,0)，4 个数字分别对应每个字符的左下角 x 坐标、左下角 y 坐标、右上角 x 坐标和右上角 y 坐标），最后一个数字“0”表示图片样本的编号。

显然，手工创建这些图片矩形定位文件很无聊，不过有一些工具可以帮你完成。我很喜欢在线工具 Tesseract OCR Chopper（http://pp19dd.com/tesseract-ocr-chopper/），因为它不需要安装，也没有其他依赖，只要有浏览器就可以运行，而且用法很简单：上传图片，如果要增加新矩形就单击“add”按钮，还可以根据需要调整矩形的尺寸，最后把新生成的矩形定位文件复制到一个新文件里就可以了。

矩形定位文件必须保存在一个 .box 后缀的文本文件中。和图片文件一样，文本文件也是用验证码的实际结果命名（例如，4MmC3.box）。另外，这样便于检查 .box 文件的内容和文

件的名称，而且按文件名对目录中的文件排序之后，就可以让 .box 文件与对应的图片文件的实际结果进行对比。

你还需要创建大约 100 个 .box 文件来保证你有足够的训练数据。因为 Tesseract 会忽略那些不能读取的文件，所以建议你尽量多做一些矩形定位文件，以保证训练足够充分。如果你觉得训练的 OCR 结果没有达到你的目标，或者 Tesseract 识别某些字符时总是出错，多创建一些训练数据然后重新训练将是一个不错的改进方法。

创建完满载 .box 文件和图片文件的数据文件夹之后，在做进一步分析之前最好备份一下这个文件夹。虽然在数据上运行训练程序不太可能删除任何数据，但是创建 .box 文件用了你好几个小时的时间，来之不易，稳妥一点儿总没错。此外，能够抓取一个满是编译数据的混乱目录，然后再尝试一次，总是好的。

完成所有的数据分析工作和创建 Tesseract 所需的训练文件，一共有六个步骤。有一些工具可以帮你处理图片和 .box 文件，不过目前 Tesseract 3.02 还不支持。

我写了一个 Python 版的解决方案（https://github.com/REMitchell/tesseract-trainer）来处理同时包含图片文件和 .box 文件的数据文件夹，然后自动创建所有必需的训练文件。

这个解决方案的主要配置方式和步骤都在 `main` 方法（目前，作者已经在 GitHub 中将示例代码修改为 `__init__` 方法，符合 Python 的类定义原则）和 `runAll` 方法里：

```
def main(self):
    languageName = "eng"
    fontName = "captchaFont"
    directory = "<path to images>"

def runAll(self):
    self.createFontFile()
    self.cleanImages()
    self.renameFiles()
    self.extractUnicode()
    self.runShapeClustering()
    self.runMfTraining()
    self.runCnTraining()
    self.createTessData()
```

你需要动手设置的只有三个变量。

- `languageName`
  Tesseract 用三个字母的语言缩写代码表示识别的语言种类。可能大多数情况下，你都会用“eng”表示英语（English）。

- `fontName`
  表示你选择的字体名称，可以是任意名称，但必须是一个不包含空格的单词。

- `directory`

  表示包含所有图片和 .box 文件的目录。建议你使用文件夹的绝对路径，但是如果你使用相对路径，可能需要以 Python 代码运行的目录位置为原点。如果你使用绝对路径，就可以在电脑的任意位置运行代码了。

让我们再看看 `runAll` 里每个函数的用法。

`createFontFile` 创建了一个 `font_properties` 文件，让 Tesseract 知道我们要创建的新字体：

```
captchaFont 0 0 0 0 0
```

这个文件包括字体的名称，后面跟着若干 1 和 0，分别表示应该使用斜体、加粗或其他版本的字体（用这些属性训练字体是一个很好玩儿的练习，不过超出了本书的介绍范围，感兴趣的同学可以自己尝试）。

`cleanImages` 首先创建所有样本图片的高对比度版本，然后转换成灰度图，并进行一些清理，让 Tesseract 更容易读取图片文件。如果你要处理的验证码图片上面有一些很容易过滤掉的噪点，那么你可以在这里增加一些步骤来处理它们。

`renameFiles` 把所有的图片文件和 .box 文件的文件名改变成 Tesseract 需要的形式（fileNumber 是文件序号，用来区别每个文件）：

- <languageName>.<fontName>.exp<fileNumber>.box
- <languageName>.<fontName>.exp<fileNumber>.tiff

`extractUnicode` 函数会检查所有已创建的 .box 文件，确定要训练的字符集范围。抽取出的 Unicode 会告诉你一共找到了多少个不重复的字符，这也是一个查询字符的好方法，如果你漏了字符可以用这个结果快速排查。

之后的三个函数，`runShapeClustering`、`runMfTraining` 和 `runCtTraining` 分别用来创建文件 `shapetable`、`pfftable` 和 `normproto`。它们会生成每个字符的几何和形状信息，也为 Tesseract 提供计算字符若干可能结果的概率统计信息。

最后，Tesseract 会用之前设置的语言名称对数据文件夹编译出的每个文件进行重命名（例如，shapetable 被重命名为 eng.shapetable），然后把所有的文件编译到最终的训练文件 eng.traineddata 中。

你需要动手完成的唯一步骤，就是用下面的 Linux 和 Mac 命令行把刚刚创建的 eng.traineddata 文件复制到 tessdata 文件夹里，Windows 系统类似：

```
$cp /path/to/data/eng.traineddata $TESSDATA_PREFIX/tessdata
```

经过这些步骤之后，你就可以用这些 Tesseract 训练过的验证码来识别新图片了。现在我们

用 Tesseract 重新读取之前的示例验证码图片，就可以得到正确的结果了：

```
$ tesseract captchaExample.png output;cat output.txt
4MmC3
```

成功啦！相比之前的识别结果“4N\,,,C<3”，这个识别结果有明显的改善。

前面的内容只是对 Tesseract 库强大的字体训练和识别能力的一个简略概述。如果你对 Tesseract 的其他训练方法感兴趣，甚至打算建立自己的验证码训练文件库，或者想和全世界的 Tesseract 爱好者分享自己对一种新字体的识别成果，那么我推荐你仔细阅读 Tesseract 的文档（https://github.com/tesseract-ocr/tesseract/wiki）。

11.4　获取验证码提交答案

许多流行的内容管理系统即使加了验证码模块，其众所周知的注册页面也经常会遭到网络机器人的垃圾注册。比如在 http://pythonscraping.com/ 上，即使加了验证码（的确也很容易识别）也不能让“汹涌澎湃”的垃圾注册有所缓解。

那么，这些网络机器人究竟是怎么做的呢？既然我们已经可以成功地识别出保存在电脑上的验证码了，那么如何才能实现一个全能的网络机器人呢？下面的示例将综合前面几章的内容来告诉你答案。如果你还没准备好，请至少浏览一下前几章关于表单提交和文件下载的相关内容。

大多数网站生成的验证码图片都具有以下属性。

- 它们是服务器端的程序动态生成的图片。验证码图片的 `src` 属性可能和普通图片不太一样，比如 `<img src="WebForm.aspx?id=8AP85CQKE9TJ">`，但是可以和其他图片一样进行下载和处理。
- 图片的答案存储在服务器端的数据库里。
- 很多验证码都有时间限制，如果你太长时间没解决就会失效。虽然这对网络机器人来说不是什么问题，但是如果你想保留验证码的答案一会儿再使用，或者想通过一些方法延长验证码的有效时限，可能很难成功。

常用的处理方法就是，首先把验证码图片下载到硬盘里，清理干净，然后用 Tesseract 处理图片，最后返回符合网站要求的识别结果。

我在 http://pythonscraping.com/humans-only 创建了一个带验证码的评论表单，演示如何用网络机器人破解验证码。程序如下所示：

```
from urllib.request import urlretrieve
from urllib.request import urlopen
from bs4 import BeautifulSoup
import subprocess
```

```
import requests
from PIL import Image
from PIL import ImageOps

def cleanImage(imagePath):
    image = Image.open(imagePath)
    image = image.point(lambda x: 0 if x<143 else 255)
    borderImage = ImageOps.expand(image, border=20, fill='white')
    borderImage.save(imagePath)

html = urlopen("http://www.pythonscraping.com/humans-only")
bsObj = BeautifulSoup(html)
# 收集需要处理的表单数据(包括验证码和输入字段)
imageLocation = bsObj.find("img", {"title": "Image CAPTCHA"})["src"]
formBuildId = bsObj.find("input", {"name":"form_build_id"})["value"]
captchaSid = bsObj.find("input", {"name":"captcha_sid"})["value"]
captchaToken = bsObj.find("input", {"name":"captcha_token"})["value"]

captchaUrl = "http://pythonscraping.com"+imageLocation
urlretrieve(captchaUrl, "captcha.jpg")
cleanImage("captcha.jpg")
p = subprocess.Popen(["tesseract", "captcha.jpg", "captcha"],stdout=
    subprocess.PIPE,stderr=subprocess.PIPE)
p.wait()
f = open("captcha.txt", "r")

# 清理识别结果中的空格和换行符
captchaResponse = f.read().replace(" ", "").replace("\n", "")
print("Captcha solution attempt: "+captchaResponse)

if len(captchaResponse) == 5:
    params = {"captcha_token":captchaToken, "captcha_sid":captchaSid,
              "form_id":"comment_node_page_form", "form_build_id": formBuildId,
                  "captcha_response":captchaResponse, "name":"Ryan Mitchell",
                  "subject": "I come to seek the Grail",
                  "comment_body[und][0][value]":
                                          "...and I am definitely not a bot"}
    r = requests.post("http://www.pythonscraping.com/comment/reply/10",
                          data=params)
    responseObj = BeautifulSoup(r.text)
    if responseObj.find("div", {"class":"messages"}) is not None:
        print(responseObj.find("div", {"class":"messages"}).get_text())
else:
    print("There was a problem reading the CAPTCHA correctly!")
```

值得注意的是，有两种异常情况会导致这个程序运行失败。第一种情况是，如果 Tesseract 从验证码图片中识别的结果不是五个字符（因为训练样本中验证码的所有有效答案都必须是五个字符），结果不会被提交，程序失败。第二种情况是虽然识别的结果是五个字符，被提交到了表单，但是服务器对结果不认可，程序仍然失败。在实际运行过程中，第一种情况发生的可能性大约为 50%，发生时程序不会向表单提交，程序直接结束并提示验证码识别错误。第二种异常情况发生的概率约为 20%，五个字符都对的概率约是 30%（每个字母的识别正确率大约是 80%，5 个字符都识别正确的总概率是 32.8%）。

虽然这个程序的识别效果好像很差，但是用户尝试填写验证码的次数并没有限制，而且大多数错误的识别结果都可以在提交到表单之前就被拦下来。因此，如果有一个识别结果提交到表单并传送到服务器，那么验证码很可能就是正确的。如果这样解释并不能让你信服，请记住这些都只是简单的猜测，准确率只有 0.0000001%。[3] 程序只要运行三到四次就可以识别出一个验证码，比简单的猜测 9 亿次还是要节省很多时间的！

注 3：验证码的字符集是 26 个大写字母，26 个小写字母，10 个数字，5 个字符一共有 52 的 5 次方，即 916 132 832 种可能，因此简单猜测的准确率只有 0.0000001%，下一句中的 9 亿次就是这个道理。
——译者注

第 12 章

避开采集陷阱

在采集网站的时候，还会遇到一些比数据显示在浏览器上却抓取不出来更令人沮丧的事情。也许是向服务器提交自认为已经处理得很好的表单却被拒绝，也许是自己的 IP 地址不知道什么原因直接被网站封杀，无法继续访问。

这是由于一些堪称最复杂的 bug 还没有解决，不仅因为这些 bug 让人意想不到（程序在一个网站上可以正常使用，但在另一个看起来完全一样的网站上却用不了），还因为那些网站有意不让爬虫抓取信息。网站已经把你定性为一个网络机器人直接拒绝，你无法找出原因。

在这本书里，我已经写了一堆方法来处理网站抓取的难点（提交表单，抽取和清理数据，执行 JavaScript，等等）。这一章将继续介绍更多的知识点，尽管属于不同的主题（HTTP headers、CSS 和 HTML 表单等），但这些知识点的共同目的都是为了克服网站阻止自动采集这个障碍。

即使你觉得下面这些内容现在对你没什么用，我还是强烈建议你至少浏览一下。也许有一天，这一章的内容会帮你解决一个非常复杂的 bug，或者防止那类 bug 发生。

12.1　道德规范

在本书前几章里，我介绍过网络数据采集行为的法律灰色地带，以及网络数据采集涉及的一些道德规范。说实话，从道德角度上说，这一章是我在写这本书时感到最困难的一章。我自己的网站已经被网络机器人、垃圾邮件生成器、网络爬虫和其他各种不受欢迎的虚拟访问者骚扰过很多次了，你的网站可能也是一样。既然如此，我为什么还要在这一章教人们建立更强大的网络机器人呢？

有几个很重要的理由促使我写这一章。

- 在采集那些不想被采集的网站时，其实存在一些非常符合道德和法律规范的理由。比如我之前的工作就是做网络爬虫，我曾做过一个自动信息收集器，从未经许可的网站上自动收集客户的名称、地址、电话号码和其他个人信息，然后把采集的信息提交到网站上，让服务器删除这些客户信息。为了避免竞争，这些网站都会对网络爬虫严防死守。但是，我的工作要确保公司的客户们都匿名（这些人都是家庭暴力受害者，或者因其他正当理由想保持低调的人），这为网络数据采集工作创造了极其合理的条件，我很高兴自己有能力从事这项工作。
- 虽然不太可能建立一个完全“防爬虫”的网站（最起码得让合法的用户可以方便地访问网站），但我还是希望本章的内容可以帮助人们保护自己的网站不被恶意攻击。在这一章的内容里，我将指出每一种网络数据采集技术的缺点，你可以利用这些缺点保护自己的网站。其实，大多数网络机器人一开始都只能做一些宽泛的信息和漏洞扫描，用本章介绍的几个简单技术就可以挡住 99% 的机器人。但是，它们进化的速度非常快，最好时刻准备迎接新的攻击。
- 和大多数程序员一样，我从来不相信禁止某一类信息的传播就可以让世界变得更和谐。

学习这一章的内容时，希望你牢记这里演示的许多程序和介绍的技术都不应该在任何一个网站上使用。不仅因为这么做对网站不好，而且你可能会收到一个停止并终止警告信（cease-and-desist letter），甚至还有可能发生更糟糕的事情。不过我也不想每次学习新技术时都警告你一下。好吧，对本书后面的内容——如哲学家阿甘曾说的——“我想说的就是这些”（That's all I have to say about that）。

12.2　让网络机器人看起来像人类用户

网站防采集的前提就是要正确地区分人类访问用户和网络机器人。虽然网站可以使用很多识别技术（比如验证码）来防止爬虫，但还是有一些十分简单的方法，可以让你的网络机器人看起来更像人类访问用户。

12.2.1　修改请求头

在第 9 章里，我们曾经用 `requests` 模块处理网站的表单。`requests` 模块还是一个设置请求头的利器。HTTP 的请求头是在你每次向网络服务器发送请求时，传递的一组属性和配置信息。HTTP 定义了十几种古怪的请求头类型，不过大多数都不常用。只有下面的七个字段被大多数浏览器用来初始化所有网络请求（表中信息是我自己浏览器的数据）。

属　　性	内　　容
`Host`	https://www.google.com/
`Connection`	keep-alive

（续）

属　　性	内　　容
Accept	text/html，application/xhtml+xml，application/xml;q=0.9，image/webp，*/*;q=0.8
User-Agent	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.95 Safari/537.36
Referrer	https://www.google.com/
Accept-Encoding	gzip，deflate，sdch
Accept-Language	en-US,en;q=0.8

经典的 Python 爬虫在使用 `urllib` 标准库时，都会发送如下的请求头：

属　　性	内　　容
Accept-Encoding	identity
User-Agent	Python-urllib/3.4

如果你是一个防范爬虫的网站管理员，你会让哪个请求头访问你的网站呢？

安装 Requests

我们在第 9 章已经安装过 Requests 模块了，如果你还没有装，可以在模块的网站上找到下载链接（http://docs.python-requests.org/en/latest/user/install/）和安装方法，或者用任意第三方 Python 模块安装器进行安装。

请求头可以通过 `requests` 模块进行自定义。https://www.whatismybrowser.com/ 网站就是一个非常棒的网站，可以让服务器测试浏览器的属性。我们用下面的程序来采集这个网站的信息，验证我们浏览器的 cookie 设置：

```
import requests
from bs4 import BeautifulSoup

session = requests.Session()
headers = {"User-Agent":"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_5)
                         AppleWebKit 537.36 (KHTML, like Gecko) Chrome",
           "Accept":"text/html,application/xhtml+xml,application/xml;
                     q=0.9,image/webp,*/*;q=0.8"}
url = "https://www.whatismybrowser.com/
       developers/what-http-headers-is-my-browser-sending"
req = session.get(url, headers=headers)

bsObj = BeautifulSoup(req.text)
print(bsObj.find("table",{"class":"table-striped"}).get_text)
```

程序输出结果中的请求头应该和程序中设置的 `headers` 是一样的。

虽然网站可能会对 HTTP 请求头的每个属性进行“是否具有人性”的检查，但是我发现通常真正重要的参数就是 `User-Agent`。无论你在做什么项目，一定要记得把 `User-Agent` 属性

设置成不容易引起怀疑的内容，不要用 Python-urllib/3.4。另外，如果你正在处理一个警觉性非常高的网站，就要注意那些经常用却很少检查的请求头，比如 Accept-Language 属性，也许它正是那个网站判断你是个人类访问者的关键。

请求头会改变你观看网络世界的方式

假设你想为一个机器学习的研究项目写一个语言翻译机，却没有大量的翻译文本来测试它的效果。很多大型网站都会为同样的内容提供不同的语言翻译，根据请求头的参数响应网站不同的语言版本。因此，你只要简单地把请求头属性从 Accept-Language:en-US 修改成 Accept-Language:fr，就可以从网站上获得“Bonjour”（法语，你好）这些数据来改善翻译机的翻译效果了（大型跨国企业通常都是好的采集对象）。

请求头还可以让网站改变内容的布局样式。例如，用移动设备浏览网站时，通常会看到一个没有广告、Flash 以及其他干扰的简化的网站版本。因此，把你的请求头 User-Agent 改成下面这样，就可以看到一个更容易采集的网站了！

> User-Agent:Mozilla/5.0 (iPhone; CPU iPhone OS 7_1_2 like Mac OS X) AppleWebKit/537.51.2 (KHTML, like Gecko) Version/7.0 Mobile/11D257 Safari/9537.53

12.2.2 处理cookie

虽然 cookie 是一把双刃剑，但正确地处理 cookie 可以避免许多采集问题。网站会用 cookie 跟踪你的访问过程，如果发现了爬虫异常行为就会中断你的访问，比如特别快速地填写表单，或者浏览大量页面。虽然这些行为可以通过关闭并重新连接或者改变 IP 地址来伪装（更多信息请参见第 14 章），但是如果 cookie 暴露了你的身份，再多努力也是白费。

在采集一些网站时 cookie 是不可或缺的。在第 9 章的例子中曾经介绍过，在一个网站上持续地保持登录状态，需要你在多个页面中保存一个 cookie。一些网站不要求在每次登录时都获得一个新 cookie，只要保存一个旧的“已登录”的 cookie 就可以访问网站。

如果你在采集一个或者几个目标网站，我建议你检查这些网站生成的 cookie，然后想想哪一个 cookie 是爬虫需要处理的。有一些浏览器插件可以为你显示访问网站和离开网站时 cookie 是如何设置的。EditThisCookie（http://www.editthiscookie.com/）就是我最喜欢的 Chrome 浏览器插件之一。

要获得 cookie 的更多信息，请查看 9.5 节，里面的示例代码介绍了使用 requests 模块处理 cookie 的过程。当然，因为 requests 模块不能执行 JavaScript，所以它不能处理很多新式的跟踪软件生成的 cookie，比如 Google Analytics，只有当客户端脚本执行后才设置 cookie（或者在用户浏览页面时基于网页事件产生 cookie，比如点击按钮）。为了处理这些动作，你需要用 Selenium 和 PhantomJS 包（基本的安装和用法在第 10 章已经介绍过）。

你可以对任意网站（本例用的是 http://pythonscraping.com）调用 webdriver 的 `get_cookie()` 方法来查看 cookie：

```
from selenium import webdriver
driver = webdriver.PhantomJS(executable_path='<Path to Phantom JS>')
driver.get("http://pythonscraping.com")
driver.implicitly_wait(1)
print(driver.get_cookies())
```

这样就可以获得一个非常典型的 Google Analytics 的 cookie 列表：

```
[{'value': '1', 'httponly': False, 'name': '_gat', 'path': '/', 'expi
ry': 1422806785, 'expires': 'Sun, 01 Feb 2015 16:06:25 GMT', 'secure'
: False, 'domain': '.pythonscraping.com'}, {'value': 'GA1.2.161952506
2.1422806186', 'httponly': False, 'name': '_ga', 'path': '/', 'expiry
': 1485878185, 'expires': 'Tue, 31 Jan 2017 15:56:25 GMT', 'secure':
False, 'domain': '.pythonscraping.com'}, {'value': '1', 'httponly': F
alse, 'name': 'has_js', 'path': '/', 'expiry': 1485878185, 'expires':
 'Tue, 31 Jan 2017 15:56:25 GMT', 'secure': False, 'domain': 'pythons
craping.com'}]
```

你还可以调用 `delete_cookie()`、`add_cookie()` 和 `delete_all_cookies()` 方法来处理 cookie。另外，还可以保存 cookie 以备其他网络爬虫使用。下面的例子演示了如何把这些函数组合在一起：

```
from selenium import webdriver

driver = webdriver.PhantomJS(executable_path='<Path to Phantom JS>')
driver.get("http://pythonscraping.com")
driver.implicitly_wait(1)
print(driver.get_cookies())

savedCookies = driver.get_cookies()

driver2 = webdriver.PhantomJS(executable_path='<Path to Phantom JS>')
driver2.get("http://pythonscraping.com")
driver2.delete_all_cookies()
for cookie in savedCookies:
    driver2.add_cookie(cookie)

driver2.get("http://pythonscraping.com")
driver.implicitly_wait(1)
print(driver2.get_cookies())
```

在这个例子中，第一个 webdriver 获得了一个网站，打印 cookie 并把它们保存到变量 `savedCookies` 里。第二个 webdriver 加载同一个网站（技术提示：必须首先加载网站，这样 Selenium 才能知道 cookie 属于哪个网站，即使加载网站的行为对我们没任何用处），删除所有的 cookie，然后替换成第一个 webdriver 得到的 cookie。当再次加载这个页面时，两组 cookie 的时间戳、源代码和其他信息应该完全一致。从 Google Analytics 的角度看，第二个 webdriver 现在和第一个 webdriver 完全一样。

12.2.3 时间就是一切

有一些防护措施完备的网站可能会阻止你快速地提交表单，或者快速地与网站进行交互。即使没有这些安全措施，用一个比普通人快很多的速度从一个网站下载大量信息也可能让自己被网站封杀。

因此，虽然多线程程序可能是一个快速加载页面的好办法——让你在一个线程中处理数据并在另一个线程中加载页面——但是这对编写好的爬虫来说依然是一个恐怖的策略。还是应该尽量保证一次加载页面加载且数据请求最小化。如果条件允许，尽量为每个页面访问增加一点儿时间间隔，即使你要增加一行代码：

```
time.sleep(3)
```

虽然网络数据采集经常会为了获取数据而破坏规则和冲破底线，但是合理控制速度是你不应该破坏的规则。这不仅是因为过度消耗别人的服务器资源会让你置身于非法境地，而且你这么做可能会把一个小型网站拖垮甚至下线。拖垮网站是一件不道德的事情：是彻头彻尾的错误。所以请控制你的采集速度！

12.3 常见表单安全措施

许多像 Litmus 之类的测试工具已经用了很多年了，现在仍用于区分网络爬虫和使用浏览器的人类访问者，这类手段都取得了不同程度的效果。虽然网络机器人下载一些公开的文章和博文并不是什么大事，但是如果网络机器人在你的网站上创造了几千个账号并开始向所有用户发送垃圾邮件，就是一个大问题了。网络表单，尤其是那些用于账号创建和登录的网站，如果被机器人肆意地滥用，网站的安全和流量费用就会面临严重威胁，因此努力限制网站的接入是最符合许多网站所有者的利益的（至少他们这么认为）。

这些集中在表单和登录环节上的反机器人安全措施，对网络爬虫来说确实是严重的挑战。

当你为那些表单创建自动化机器人时，你会遇到的安全措施不止这些。关于处理表单的更多信息，请参考第 11 章关于处理验证码和图片处理的内容，以及第 14 章关于请求头和 IP 地址处理的内容。

12.3.1 隐含输入字段值

在 HTML 表单中，“隐含”字段可以让字段的值对浏览器可见，但是对用户不可见（除非看网页源代码）。随着越来越多的网站开始用 cookie 存储状态变量来管理用户状态，在找到另一个最佳用途之前，隐含字段主要用于阻止爬虫自动提交表单。

图 12-1 显示的例子就是 Facebook 登录页面上的隐含字段。虽然表单里只有三个可见字段（username、password 和一个确认按钮），但是在源代码里表单会向服务器传送大量的信息。



图 12-1：Facebook 登录页面上的隐含字段

用隐含字段阻止网络数据采集的方式主要有两种。第一种是表单页面上的一个字段可以用服务器生成的随机变量表示。如果提交时这个值不在表单处理页面上，服务器就有理由认为这个提交不是从原始表单页面上提交的，而是由一个网络机器人直接提交到表单处理页面的。绕开这个问题的最佳方法就是，首先采集表单所在页面上生成的随机变量，然后再提交到表单处理页面。

第二种方式是“蜜罐”（honey pot）。如果表单里包含一个具有普通名称的隐含字段（设置蜜罐圈套），比如“用户名”（username）或“邮箱地址”（email address），设计不太好的网络机器人往往不管这个字段是不是对用户可见，直接填写这个字段并向服务器提交，这样就会中服务器的蜜罐圈套。服务器会把所有隐含字段的真实值（或者与表单提交页面的默认值不同的值）都忽略，而且填写隐含字段的访问用户也可能被网站封杀。

总之，有时检查表单所在的页面十分必要，看看有没有遗漏或弄错一些服务器预先设定好的隐含字段（蜜罐圈套）。如果你看到一些隐含字段，通常带有较大的随机字符串变量，那么很可能网络服务器会在表单提交的时候检查它们。另外，还有其他一些检查，用来保证这些当前生成的表单变量只被使用一次或是最近生成的（这样可以避免变量被简单地存储到一个程序中反复使用）。

12.3.2 避免蜜罐

虽然在进行网络数据采集时用 CSS 属性区分有用信息和无用信息会很容易（比如，通过读取 `id` 和 `class` 标签获取信息），但这么做有时也会出问题。如果网络表单的一个字段通过 CSS 设置成对用户不可见，那么可以认为普通用户访问网站的时候不能填写这个字段，因为它没有显示在浏览器上。如果这个字段被填写了，就可能是机器人干的，因此这个提交会失效。

这种手段不仅可以应用在网站的表单上，还可以应用在链接、图片、文件，以及一些可以被机器人读取，但普通用户在浏览器上却看不到的任何内容上面。访问者如果访问了网站

上的一个“隐含”内容，就会触发服务器脚本封杀这个用户的 IP 地址，把这个用户踢出网站，或者采取其他措施禁止这个用户接入网站。实际上，许多商业模式就是在干这些事情。

下面的例子所用的网页在 http://pythonscraping.com/pages/itsatrap.html。这个页面包含了两个链接，一个通过 CSS 隐含了，另一个是可见的。另外，页面上还包括两个隐含字段：

```
<html>
<head>
    <title>A bot-proof form</title>
</head>
<style>
    body {
        overflow-x:hidden;
    }
    .customHidden {
        position:absolute;
        right:50000px;
    }
</style>
<body>
    <h2>A bot-proof form</h2>
    <a href=
     "http://pythonscraping.com/dontgohere" style="display:none;">Go here!</a>
    <a href="http://pythonscraping.com">Click me!</a>
    <form>
        <input type="hidden" name="phone" value="valueShouldNotBeModified"/><p/>
        <input type="text" name="email" class="customHidden"
                  value="intentionallyBlank"/><p/>
        <input type="text" name="firstName"/><p/>
        <input type="text" name="lastName"/><p/>
        <input type="submit" value="Submit"/><p/>
    </form>
</body>
</html>
```

这三个元素通过三种不同的方式对用户隐藏：

- 第一个链接是通过简单的 CSS 属性设置 `display:none` 进行隐藏
- 电话号码字段 `name="phone"` 是一个隐含的输入字段
- 邮箱地址字段 `name="email"` 是将元素向右移动 50 000 像素（应该会超出电脑显示器的边界）并隐藏滚动条

因为 Selenium 可以获取访问页面的内容，所以它可以区分页面上的可见元素与隐含元素。通过 `is_displayed()` 可以判断元素在页面上是否可见。

例如，下面的代码示例就是获取前面那个页面的内容，然后查找隐含链接和隐含输入字段：

```
from selenium import webdriver
from selenium.webdriver.remote.webelement import WebElement
```

```
driver = webdriver.PhantomJS(executable_path='')
driver.get("http://pythonscraping.com/pages/itsatrap.html")
links = driver.find_elements_by_tag_name("a")
for link in links:
    if not link.is_displayed():
        print("The link "+link.get_attribute("href")+" is a trap")

fields = driver.find_elements_by_tag_name("input")
for field in fields:
    if not field.is_displayed():
        print("Do not change value of "+field.get_attribute("name"))
```

Selenium 抓取出了每个隐含的链接和字段，结果如下所示：

```
The link http://pythonscraping.com/dontgohere is a trap
Do not change value of phone
Do not change value of email
```

虽然你不太可能会去访问你找到的那些隐含链接，但是在提交前，记得确认一下那些已经在表单中、准备提交的隐含字段的值（或者让 Selenium 为你自动提交）。

12.4 问题检查表

这一章介绍的大量知识，其实和这本书一样，都是在介绍如何建立一个更像人而不是更像机器人的网络爬虫。如果你一直被网站封杀却找不到原因，那么这里有个检查列表，可以帮你诊断一下问题出在哪里。

- 首先，如果你从网络服务器收到的页面是空白的，缺少信息，或其遇到他不符合你预期的情况（或者不是你在浏览器上看到的内容），有可能是因为网站创建页面的 JavaScript 执行有问题。可以看看第 10 章内容。
- 如果你准备向网站提交表单或发出 `POST` 请求，记得检查一下页面的内容，看看你想提交的每个字段是不是都已经填好，而且格式也正确。用 Chrome 浏览器的网络面板（快捷键 F12 打开开发者控制台，然后点击"Network"即可看到）查看发送到网站的 `POST` 命令，确认你的每个参数都是正确的。
- 如果你已经登录网站却不能保持登录状态，或者网站上出现了其他的"登录状态"异常，请检查你的 cookie。确认在加载每个页面时 cookie 都被正确调用，而且你的 cookie 在每次发起请求时都发送到了网站上。
- 如果你在客户端遇到了 HTTP 错误，尤其是 403 禁止访问错误，这可能说明网站已经把你的 IP 当作机器人了，不再接受你的任何请求。你要么等待你的 IP 地址从网站黑名单里移除，要么就换个 IP 地址（可以去星巴克上网，或者看看第 14 章的内容）。如果你确定自己并没有被封杀，那么再检查下面的内容。
  - 确认你的爬虫在网站上的速度不是特别快。快速采集是一种恶习，会对网管的服务器造成沉重的负担，还会让你陷入违法境地，也是 IP 被网站列入黑名单的首要原因。

给你的爬虫增加延迟，让它们在夜深人静的时候运行。切记：匆匆忙忙写程序或收集数据都是拙劣项目管理的表现；应该提前做好计划，避免临阵慌乱。

- 还有一件必须做的事情：修改你的请求头！有些网站会封杀任何声称自己是爬虫的访问者。如果你不确定请求头的值怎样才算合适，就用你自己浏览器的请求头吧。
- 确认你没有点击或访问任何人类用户通常不能点击或接入的信息（更多信息请查阅 12.3.2 节）。
- 如果你用了一大堆复杂的手段才接入网站，考虑联系一下网管吧，告诉他们你的目的。试试发邮件到 webmaster@< 域名 > 或 admin@< 域名 >，请求网管允许你使用爬虫采集数据。管理员也是人嘛！

第 13 章

用爬虫测试网站

当研发一个技术栈较大的网络项目时，经常只对栈底（项目后期用的技术）进行一些常规测试。目前大多数编程语言（包括 Python）都有一些测试框架，但是网站的前端通常并没有自动化测试，尽管前端才是整个项目中真正与用户零距离接触的唯一一个部分。

部分原因是网站常常是不同标记语言和编程语言的大杂烩。你可以为 JavaScript 部分写单元测试，但没什么用，如果 JavaScript 交互的 HTML 内容改变了，那么即使 JavaScript 可以正常地运行，也不能完成网页需要的动作。

网站前端测试经常被当作一件放到最后才做的事情，或者指派给低级程序员去做，最多再给他们一个检查表和一个 bug 跟踪器。但其实只要再稍微努点儿力，我们就可以把检查表变成单元测试，用网络爬虫代替人眼进行测试。

想象有一个由测试驱动的网络开发项目。每天进行测试以保证网络接口的每个环节的功能都是正常的。每当有新的特性加入网站，或者一个元素的位置改变时，就执行一组自动化测试。在这一章里，我将介绍测试的基础知识，以及如何用 Python 网络爬虫测试各种简单或复杂的网站。

13.1 测试简介

如果你以前从来没有为你的代码写过测试，那么现在开始再合适不过了。运行一套测试方法能够保证你的代码按照既定的目标运行，不仅可以节约你的时间，减少你对 bug 的忧虑，还可以让新版本升级变得更加简单。

什么是单元测试

测试和单元测试（unit test）这两个词基本可以看成是等价的。一般当程序员们说起“写测试”时，他们真正的意思就是“写单元测试”。而一些程序员提到写单元测试时，他们写的就是某一种测试。

虽然不同公司的单元测试定义和实践方法大相径庭，但是一个单元测试通常包含以下特点。

- 每个单元测试用于测试一个零件（component）功能的一个方面。例如，如果从银行账户取出一笔金额为负数的美元，那么单元测试就要对负数抛出适当的错误信息。

  通常，一个零件的所有单元测试都集成在同一个类（class）里。你可能有一个测试是针对从银行账户取出一笔金额为负数的美元，另一个测试是针对透支银行账户行为的单元测试。
- 每个单元测试都可以完全独立地运行，一个单元测试需要的所有启动（setup）和卸载（teardown）都必须通过这个单元测试本身去处理。单元测试不能对其他测试造成干扰，而且不论按何种顺序排列，它们都必须能够正常地运行。
- 每个单元测试通常至少包含一个断言（assertion）。例如，一个单元测试可以判断 2+2 的和是 4。有时，一个单元测试也许只包含一个失败状态（failure state）。例如，有这样一个单元测试，如果一个异常没有被抛出，测试失败；如果每个异常都顺利抛出，测试通过。
- 单元测试与生产代码是分离的。虽然它们需要导入然后在待测试的代码中使用，但是它们一般被保留在独立的类和目录中。

尽管有很多测试类型可以大写特写，像整体测试（integration test）和有效性测试（validation test）等，但本章只重点介绍单元测试。这不仅仅是因为单元测试在当前的测试驱动开发中十分主流，还因为其代码长度和灵活性使它们非常适合做示例。另外 Python 自带单元测试标准库，我们下一节就来介绍它。

13.2 Python单元测试

Python 的单元测试模块 `unittest`，所有标准版 Python 安装后都有。只要先导入模块然后继承 `unittest.TestCase` 类，就可以实现下面的功能：

- 为每个单元测试的开始和结束提供 `setUp` 和 `tearDown` 函数
- 提供不同类型的“断言”语句让测试成功或失败
- 把所有以 `test_` 开头的函数当作单元测试运行，忽略不带 `test_` 的函数

下面的例子演示了如何用 Python 实现一个非常简单的单元测试，测试 2+2=4：

```
import unittest

class TestAddition(unittest.TestCase):
    def setUp(self):
        print("Setting up the test")

    def tearDown(self):
        print("Tearing down the test")

    def test_twoPlusTwo(self):
        total = 2+2
        self.assertEqual(4, total)

if __name__ == '__main__':
    unittest.main()
```

虽然 `setUp` 和 `tearDown` 函数在这里并没有实现可用的功能，但是仍然达到了演示的目的。需要注意的是，这两个函数在每个测试的开始和结束都会运行一次，而不是把类中所有测试作为一个整体在开始或结束时各运行一次。

测试维基百科

将 Python 的 `unittest` 库与网络爬虫组合起来，就可以实现简单的网站前端功能测试（除了 JavaScript 测试，后面我们会介绍）。

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
import unittest

class TestWikipedia(unittest.TestCase):
    bsObj = None
    def setUpClass():
        global bsObj
        url = "http://en.wikipedia.org/wiki/Monty_Python"
        bsObj = BeautifulSoup(urlopen(url))

    def test_titleText(self):
        global bsObj
        pageTitle = bsObj.find("h1").get_text()
        self.assertEqual("Monty Python", pageTitle)

    def test_contentExists(self):
        global bsObj
        content = bsObj.find("div",{"id":"mw-content-text"})
        self.assertIsNotNone(content)

if __name__ == '__main__':
    unittest.main()
```

这里有两个测试：第一个是测试页面的标题是否为“Monty Python”，另一个是测试页面是否有一个 `div` 节点 `id` 属性是 `"mw-content-text"`。

需要注意的是，这个页面的内容只加载一次，全局对象 bsObj 由多个测试共享。这是通过 unittest 类的函数 setUpClass 来实现的，这个函数只在类的初始化阶段运行一次（与每个测试启动时都运行的 setUp 函数不同）。用 setUpClass 代替 setUp 可以省去不必要的页面加载；我们可以一次性采集全部内容，供多个测试使用。

虽然一次只测试一个页面看起来可能不够强大，也没什么意思，但是如果你还记得第 3 章的内容，就可以轻松地创建一个网络爬虫去遍历网站中所有的页面。下面我们来看看，当把网络爬虫和一个向页面内容添加断言的单元测试组合起来时，会发生什么呢？

有很多方法可以重复执行一个测试，但是我们必须对即将在页面上运行的所有测试都时刻保持谨慎，因为我们只加载一次页面，而且我们必须避免在内存中一次性加入大量的信息。具体设置如下所示：

```
class TestWikipedia(unittest.TestCase):
    bsObj = None
    url = None

    def test_PageProperties(self):
        global bsObj
        global url

        url = "http://en.wikipedia.org/wiki/Monty_Python"
        # 测试遇到的前100个页面
        for i in range(1, 100):
            bsObj = BeautifulSoup(urlopen(url))
            titles = self.titleMatchesURL()
            self.assertEquals(titles[0], titles[1])
            self.assertTrue(self.contentExists())
            url = self.getNextLink()
        print("Done!")

    def titleMatchesURL(self):
        global bsObj
        global url
        pageTitle = bsObj.find("h1").get_text()
        urlTitle = url[(url.index("/wiki/")+6):]
        urlTitle = urlTitle.replace("_", " ")
        urlTitle = unquote(urlTitle)
        return [pageTitle.lower(), urlTitle.lower()]

    def contentExists(self):
        global bsObj
        content = bsObj.find("div",{"id":"mw-content-text"})
        if content is not None:
            return True
        return False

    def getNextLink(self):
        #使用第5章介绍的方法返回随机链接
```

```
if __name__ == '__main__':
    unittest.main()
```

有几个地方需要注意。首先，这个类里面实际上只有一个测试。其他的函数其实都是功能性的辅助函数（helper function），即使它们做了大量计算来判断测试是否通过。因为测试函数 `test_PageProperties` 使用了断言语句，所以测试的结果最终会传到这些断言所在的函数里。

另外，`contentExists` 返回的是布尔变量，`titleMatchesURL` 返回的是字符串列表。我们之所以用列表而不用布尔变量作为断言语句的值，其原因如下所示：

```
======================================================================
FAIL: test_PageProperties (__main__.TestWikipedia)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "15-3.py", line 22, in test_PageProperties
    self.assertTrue(self.titleMatchesURL())
AssertionError: False is not true
```

`assertEquals` 语句的结果是：

```
======================================================================
FAIL: test_PageProperties (__main__.TestWikipedia)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "15-3.py", line 23, in test_PageProperties
    self.assertEquals(titles[0], titles[1])
AssertionError: 'lockheed u-2' != 'u-2 spy plane'
```

究竟哪一种方式调试起来更方便呢？

在这个示例中，之所生会发生错误是因为网页发生了一个重定向，即词条 https://en.wikipedia.org/wiki/U-2_spy_plane 会跳转到标题为“Lockheed U-2”的词条。

13.3 Selenium单元测试

和第 10 章里介绍的 Ajax 采集一样，在网站测试中 JavaScript 也是一个难题。幸运的是，我们有 Selenium，它是一个可以解决网站上各种复杂问题的优秀测试框架；其实，它的初衷就是用来做网站测试！

虽然这里的单元测试都是同一种语言（Python）写的，但是 Python 单元测试和 Selenium 单元测试的语法还是有点儿不一样。Selenium 不要求单元测试必须是类的一个函数，它的“断言”语句也不需要括号，而且测试通过的话不会有提示，只有当测试失败时才会产生信息提示：

```
driver = webdriver.PhantomJS()
```

```
driver.get("http://en.wikipedia.org/wiki/Monty_Python")
assert "Monty Python" in driver.title
driver.close()
```

当这段代码运行的时候，测试结果不会输出任何信息。

因此，写 Selenium 单元测试的时候需要比写 Python 单元测试更加随意，断言语句甚至可以整合到生产代码中，非常适合某些条件不能满足就中断代码的需求。

与网站进行交互

最近，我想通过一个网站的商户通讯录联系我家附近的一个小商家，结果发现网页上的信息没有了；我点击提交按钮的时候什么也没查到。经过一些探索之后，我发现这个网站用了一个简易的邮件发送表单，如果商户联系方式的内容有问题就可以给网管发邮件。于是我就用这个邮箱地址给他们发了一封邮件，告诉他们联系方式信息表单出了问题，让他们尽快解决，虽然不是技术问题。

如果我要写一个普通的爬虫来采集或测试这个表单，那么爬虫也许只能复制表单的结构，然后直接给我自己发邮件——不过抓不到表单的内容。那么我怎么测试表单的功能才能保证它在浏览器上也可以正常工作呢？

虽然在前面的几章中我们介绍过链接跳转、表单提交和其他网站交互行为，但是我们做那些事情的共同初衷都是要*避开*浏览器图形界面，而不是使用浏览器。另一方面，Selenium 可以在浏览器（这里用 PhantomJS 无头浏览器）上做任何事，包括输入文字、点击按钮等，这样就可以找出异常表单、JavaScript 代码错误、HTML 排版错误，以及其他用户使用过程中可能出现的问题。

这个测试的关键是使用 Selenium 的 `elements`。这个对象在第 10 章已经简单介绍过了，它的调用方式如下所示：

```
usernameField = driver.find_element_by_name('username')
```

就像你可以在浏览器里对网站上的不同元素执行一系列操作一样，Selenium 也可以对任何给定元素执行很多操作，如下所示：

```
myElement.click()
myElement.click_and_hold()
myElement.release()
myElement.double_click()
myElement.send_keys_to_element("content to enter")
```

为了一次性完成一个元素的多个操作，可以用*动作链*（action chain）储存多个操作，然后在一个程序中执行一次或多次。用动作链储存多个操作非常方便，而且非常有用，它们的功能和前面示例中对一个元素显式调用操作是完全一样的。

为了演示两种方式的差异，我们看一看 http://pythonscraping.com/pages/files/form.html 的表单（是第 9 章用过的例子）。我们用下面的方式填写表单并提交：

```
from selenium import webdriver
from selenium.webdriver.remote.webelement import WebElement
from selenium.webdriver.common.keys import Keys
from selenium.webdriver import ActionChains


driver = webdriver.PhantomJS(executable_path='<Path to Phantom JS>')
driver.get("http://pythonscraping.com/pages/files/form.html")

firstnameField = driver.find_element_by_name("firstname")
lastnameField = driver.find_element_by_name("lastname")
submitButton = driver.find_element_by_id("submit")

### 方法1 ###
firstnameField.send_keys("Ryan")
lastnameField.send_keys("Mitchell")
submitButton.click()
################

### 方法2 ###
actions = ActionChains(driver).click(firstnameField).send_keys("Ryan")
                              .click(lastnameField).send_keys("Mitchell")
                              .send_keys(Keys.RETURN)
actions.perform()
################

print(driver.find_element_by_tag_name("body").text)

driver.close()
```

方法 1 在两个字段上调用 `send_keys`，然后点击确认按钮；而方法 2 在用一个动作链来点击每个字段并填写内容，最后确认，这些行为是在 `perform` 调用之后才发生的。无论用第一个方法还是第二个方法，这个程序的结果都一样：

```
Hello there, Ryan Mitchell!
```

两个方法除了处理命令的对象不同之外，第二个方法还有一点差异：注意第一个方法提交表单是点击“确认”按钮，而第二个方法提交表单是用回车键（`Keys.RETURN`）。因为实现同样效果的网络事件发生顺序可以有多种，所以用 Selenium 实现同样的结果也有许多方式。

1. 鼠标拖放动作

单击按钮和输入文字只是 Selenium 的一个功能，其真正的亮点是能够处理更加复杂的网络表单交互行为。Selenium 可以轻松地完成鼠标拖放动作（drag-and-drop）。使用它的拖放函数，你需要指定一个被拖放的元素以及拖放的距离，或者元素将被拖放到的目标元素。

下面的例子用 http://pythonscraping.com/pages/javascript/draggableDemo.html 页面演示了拖

放动作：

```
from selenium import webdriver
from selenium.webdriver.remote.webelement import WebElement
from selenium.webdriver import ActionChains

driver = webdriver.PhantomJS(executable_path='<Path to Phantom JS>')
driver.get('http://pythonscraping.com/pages/javascript/draggableDemo.html')

print(driver.find_element_by_id("message").text)

element = driver.find_element_by_id("draggable")
target = driver.find_element_by_id("div2")
actions = ActionChains(driver)
actions.drag_and_drop(element, target).perform()

print(driver.find_element_by_id("message").text)
```

示例页面的 message 节点上显示了两条信息。第一条是：

```
Prove you are not a bot, by dragging the square from the blue area to the red
area!
```

然后任务很快就会完成，第二条内容就被打印出来：

```
You are definitely not a bot!
```

当然，就像示例页面上显示的，很多验证码里使用拖放动作证明访问者不是一个机器人，这是一种常用手段。虽然机器人也可以长时间拖着一个元素不放（就是点击，拖住，移动），但是也不知道为什么，用“拖住不放”来检验一个用户是不是机器人的方式仍然存在。

另外，这些可拖放的验证码库很少使用那些“能够难住机器人”的任务，比如“拖动小猫图片放到奶牛图片的上面”（这需要你能够识别“小猫”和“奶牛”图片）；相反，它们经常用数字排序或其他一些非常不起眼的任务，就像前面例子里的拖放。

当然，这些验证码库的优势在于那些简单任务可以实现大量的变化，而且每种变化的使用频率都不高——另外也不会有人愿意花时间去做一个能够搞定所有任务的机器人。至少这个例子可以解释为什么你不应该在大型网站上用这种技术。

2. 截屏

除了普通的测试功能，Selenium 还有一个有趣的技巧可以让你的测试更容易（或者让你老板更喜欢）：截屏。截屏可以在单元测试中创建，不需要点击截屏按钮就可以获取：

```
driver = webdriver.PhantomJS()
driver.get('http://www.pythonscraping.com/')
driver.get_screenshot_as_file('tmp/pythonscraping.png')
```

这段脚本会访问 http://pythonscraping.com/，并将主页的屏幕截图保存在本地的 tmp 文件夹中（该文件夹必须创建好，以供正确存储之用）。截屏可保存为多种文件格式。

13.4 Python单元测试与Selenium单元测试的选择

Python 的单元测试语法严谨冗长，更适合为大多数大型项目写测试，而 Selenium 的测试方式灵活且功能强大，可以成为一些网站功能测试的首选。那么应该使用哪个呢？

答案是：不需要选择。Selenium 可以轻易地获取网站的信息，而单元测试可以评估这些信息是否满足通过测试的条件。因此，你没有理由拒绝把 Selenium 导入 Python 的单元测试，两者组合是最佳拍档。

例如，下面的程序创建了一个带拖放动作的网站单元测试，如果一个元素被正确地拖放到另一个元素里，那么推断条件成立，会显示“你不是一个机器人！”（You are not a bot!），测试通过。

```
from selenium import webdriver
from selenium.webdriver.remote.webelement import WebElement
from selenium.webdriver import ActionChains
import unittest

class TestAddition(unittest.TestCase):
    driver = None
    def setUp(self):
        global driver
        driver = webdriver.PhantomJS(executable_path='<Path to Phantom JS>')
        url = 'http://pythonscraping.com/pages/javascript/draggableDemo.html'
        driver.get(url)

    def tearDown(self):
        print("Tearing down the test")

    def test_drag(self):
        global driver
        element = driver.find_element_by_id("draggable")
        target = driver.find_element_by_id("div2")
        actions = ActionChains(driver)
        actions.drag_and_drop(element, target).perform()

        self.assertEqual("You are definitely
                          not a bot!", driver.find_element_by_id(
                          "message").text)

if __name__ == '__main__':
    unittest.main()
```

任何网站上可以看到的内容都可以通过 Python 的单元测试和 Selenium 组合来测试。其实，如果再与第 11 章介绍的一些图像处理库组合起来，就可以通过网站截屏实现像素级测试了！

第 14 章

远程采集

本章内容放在最后来介绍还是比较合适的。到现在为止，我们已经在自己的电脑上通过命令行运行了所有的 Pyhton 程序。当然，你可能也安装了 MySQL，尝试真实的服务器环境。但是这和实际的服务器还是不一样的。正如一句谚语所说："如果你喜欢某个东西，就放开手。"

这一章我们将介绍几种方法，让程序在不同的机器上运行，或者在你的电脑上用不同的 IP 地址运行。你可能打算放弃这一章，因为你现在还不需要这些内容，但是你可能会感到惊讶，自己原来已经拥有非常容易上手的工具了（比如一些付费的 VPS 或云计算资源），而且当你停止在自己的笔记本上运行 Python 爬虫后，生活会变得更加轻松。

14.1　为什么要用远程服务器

虽然使用远程服务器看起来可能更像是在启动一个供广大用户使用的网络应用时所采取的必然步骤，但我们为个人目的建立的工具通常都必须在本地运行。启用远程平台的人通常有两个目的：对更大计算能力和灵活性的需求，以及对可变 IP 地址的需求。

14.1.1　避免IP地址被封杀

建立网络爬虫的第一原则是：所有信息都可以伪造。你可以用非本人的邮箱发送邮件，通过命令行自动化鼠标的行为，或者通过 IE5.0 浏览器耗费网站流量来吓唬网管。

但是有一件事情是不能作假的，那就是你的 IP 地址。任何人都可以用这个地址给你写信："美国华盛顿特区宾夕法尼亚大道西北 1600 号，总统，邮编 20500。"但是，如果这封信是

从新墨西哥州的阿尔伯克基市发来的，那么你肯定可以确信给你写信的不是美国总统。[1]

阻止网站被采集的注意力主要集中在识别人类与机器人的行为差异上面。封杀 IP 地址这种矫枉过正的行为，就好像是农民不靠喷农药给庄稼杀虫，而是直接用火烧彻底解决问题。它是最后一步棋，不过是一种非常有效的方法，只要忽略危险 IP 地址发来的数据包就可以了。但是，使用这种方法会遇到以下几个问题。

- IP 地址访问列表很难维护。虽然大多数大型网站都会用自己的程序自动管理 IP 地址访问列表（机器人封杀机器人），但是至少需要人偶尔检查一下列表，或者至少要监控问题的增长。
- 因为服务器需要根据 IP 地址访问列表去检查每个准备接收的数据包，所以检查接收数据包时会额外增加一些处理时间。多个 IP 地址乘以海量的数据包更会使检查时间指数级增长。为了降低处理时间和处理复杂度，管理员通常会对 IP 地址进行分组管理并制定相应的规则，比如如果这组 IP 中有一些危险分子就“把这个区间的所有 256 个地址全部封杀”。于是产生了下一个问题。
- 封杀 IP 地址可能会导致意外后果。例如，当我还在美国麻省欧林工程学院读本科的时候，有个同学写了一个可以在 http://digg.com/ 网站（在 Reddit 流行之前大家都用 Digg）上对热门内容进行投票的软件。这个软件的服务器 IP 地址被 Digg 封杀，导致整个网站都不能访问。于是这个同学就把软件移到了另一个服务器上，而 Digg 自己却失去了许多主要目标用户的访问量。

虽然有这些缺点，但封杀 IP 地址依然是一种十分常用的手段，服务器管理员用它来阻止可疑的网络爬虫入侵服务器。

14.1.2 移植性与扩展性

有一些任务想通过个人电脑连网完成会十分困难。即使你并不想给任何一个网站增加负载，但是如果你会从一堆网站里收集数据，也会需要更快的网速以及更多存储空间。

另外，自己电脑上的计算资源释放之后，你就可以做很多更重要事情啦（玩魔兽，看电影，LOL）。你也不用担心电费和网速了（在星巴克启动你的应用，合上笔记本离开，每件事情都可以安全地运行），你也可以在任何有网络连接的地方收集数据。

如果你的一个应用需要非常大的计算能力，亚马逊 AWS 的一个超大计算实例也不能满足你的需求，那么你可以看看*分布式计算*（distributed computing）。这种方法可以让多个机器并发执行来完成你的任务。一个简单的例子是你可以用一台机器来采集一些网站，再用另

注 1：从技术上说，IP 地址是可以通过发送数据包进行伪装的，就是分布式拒绝服务攻击技术（Distributed Denial of Service，DDoS），攻击者不需要关心接收的数据包（这样发送请求的时候就可以使用假 IP 地址）。但是网络数据采集是一种需要关心服务器响应的行为，所以我们认为 IP 地址是不能造假的。

一台机器采集另一些网站，最后在把所有的结果存储在同一个数据库里。

当然，前几章的例子很多都是在重复 Google 搜索干的事情，但是没有几个程序可以达到 Google 搜索的运行规模。分布式计算是计算机科学中的一个庞大领域，超出了本书的介绍范围。但是，学习如何让你的程序在远程服务器执行是基本前提，学会之后你一定对当今计算机的能力感到无比惊讶。

14.2 Tor代理服务器

洋葱路由（The Onion Router）网络，常用缩写为 Tor，是一种 IP 地址匿名手段。由网络志愿者服务器构建的洋葱路由器网络，通过不同服务器构成多个层（就像洋葱）把客户端包在最里面。数据进入网络之前会被加密，因此任何服务器都不能偷取通信数据。另外，虽然每一个服务器的入站和出站通信都可以被查到，但是要想查出通信的真正起点和终点，必须知道整个通信链路上所有服务器的入站和出站通信细节，而这基本是不可能实现的。

Tor 是人权工作者和政治避难人员与记者通信的常用手段，得到了美国政府的大力支持。当然，它经常也被用于非法活动，所以也是政府盯防的目标（虽然目前盯防得并不是很成功）。

Tor 匿名的局限性

虽然我们在本书中用 Tor 的目的是改变 IP 地址，而不是实现完全匿名，但有必要关注一下 Tor 匿名方法的能力和不足。

虽然 Tor 网络可以让你访问网站时显示的 IP 地址是一个不能跟踪到你的 IP 地址，但是你在网站上留给服务器的任何信息都会暴露你的身份。例如，你登录 Gmail 账号后再用 Google 搜索，那些搜索历史就会和你的身份绑定在一起。

另外，登录 Tor 的行为也可能让你的匿名状态处于危险之中。2013 年 12 月，一个哈佛大学本科生想逃避期末考试，就用一个匿名邮箱账号通过 Tor 网络给学校发了一封炸弹威胁信。结果哈佛大学的 IT 部门通过日志查到，在炸弹威胁信发来的时候，Tor 网络的流量只来自一台机器，而且是一个在校学生注册的。虽然他们不能确定流量的最初源头（只知道是通过 Tor 发送的），但是作案时间和注册信息证据充分，而且那个时间段内只有一台机器是登录状态，这就有充分理由起诉那个学生了。

登录 Tor 网络不是一个自动的匿名措施，也不能让你进入互联网上任何区域。虽然它是一个实用的工具，但是用它的时候一定要谨慎、清醒，并且遵守道德规范。

在 Python 里使用 Tor，需要先安装运行 Tor，下一节将介绍。Tor 服务很容易安装和开启。只要去 Tor 下载页面（https://www.torproject.org/download/download）下载并安装，打开后连接就可以。不过要注意，当你用 Tor 的时候网速会变慢。这是因为代理有可能要先在全世界网络上转几次才到目的地！

PySocks

PySocks 是一个非常简单的 Python 代理服务器通信模块，它可以和 Tor 配合使用。你可以从它的网站（https://pypi.python.org/pypi/PySocks）上下载，或者使用任何第三方模块管理器安装。

这个模块的用法很简单。示例代码如下所示。运行的时候，Tor 服务必须运行在 9150 端口（默认值）上：

```
import socks
import socket
from urllib.request import urlopen

socks.set_default_proxy(socks.SOCKS5, "localhost", 9150)
socket.socket = socks.socksocket
print(urlopen('http://icanhazip.com').read())
```

网站 http://icanhazip.com/ 会显示客户端连接的网站服务器的 IP 地址，可以用来测试 Tor 是否正常运行。当程序执行之后，显示的 IP 地址就不是你原来的 IP 了。

如果你想在 Tor 里面用 Selenium 和 PhantomJS，不需要 PySocks，只要保证 Tor 在运行，然后增加 `service_args` 参数设置代理端口，让 Selenium 通过端口 9150 连接网站就可以了：

```
from selenium import webdriver
service_args = [ '--proxy=localhost:9150', '--proxy-type=socks5', ]
driver = webdriver.PhantomJS(executable_path='<path to PhantomJS>',
                             service_args=service_args)

driver.get("http://icanhazip.com")
print(driver.page_source)
driver.close()
```

和之前一样，这个程序打印的 IP 地址也不是你原来的，而是你通过 Tor 客户端获得的 IP 地址。

14.3　远程主机

一旦你使用信用卡，完全的匿名效果就消失了，即便如此，你还是可以把网络爬虫放在远程主机（Remote Hosting）上动态地改善它们的运行速度。这是因为你不仅可以自由购买服务器的使用时间，使用更强大的机器，而且网络连接也不需要在到达访问目的地之前在

Tor 网络中长途跋涉。

14.3.1　从网站主机运行

如果你拥有个人网站或公司网站，那么你可能已经知道如何使用外部服务器运行你的网络爬虫了。即使是一些相对封闭的网络服务器，没有可用的命令行接入方式，你也可以通过网页界面对程序进行控制。

如果你的网站部署在 Linux 服务器上，应该已经运行了 Python。如果你用的是 Windows 服务器，可能就没那么幸运了；你需要仔细检查一下 Python 有没有安装，或者问问网管可不可以安装。

大多数小型网络主机都会提供一个软件叫 cPanel，提供网站管理和后台服务的基本管理功能和信息。如果你接入了 cPanel，就可以设置 Python 在服务器上运行——进入“Apache Handlers”然后增加一个 handler（如还没有的话）：

```
Handler: cgi-script
Extension(s): .py
```

这会告诉服务器所有的 Python 脚本都将作为一个 CGI 脚本运行。CGI 就是通用网关接口（Common Gateway Interface），是可以在服务器上运行的任何程序，会动态地生成内容并显示在网站上。把 Python 脚本显式地定义成 CGI 脚本，就是给服务器权限去执行 Python 脚本，而不只是在浏览器上显示它们或者让用户下载它们。

写完 Python 脚本后上传到服务器，然后把文件权限设置成 755，让它可执行。通过浏览器找到程序上传的位置（也可以写一个爬虫来自动做这件事情）就可以执行程序。如果你担心在公共领域执行脚本不安全，可以采取以下两种方法。

- 把脚本存储在一个隐晦或深层的 URL 里，确保其他 URL 链接都不能接入这个脚本，这样可以避免搜索引擎发现它。
- 用密码保护脚本，或者在执行脚本之前用密码或加密令牌进行确认。

确实，通过这些原本主要是用来显示网站的服务运行 Python 脚本有点儿复杂。比如，你可能会发现网络爬虫运行时网站的加载速度变慢了。其实，在整个采集任务完成之前页面都是不会加载的（得等到所有“print”语句的输出内容都显示完）。这可能会消耗几分钟，几小时，甚至永远也完成不了，要看程序的具体情况了。虽然它最终一定能完成任务，但是可能你还想看到实时的结果，这样就需要一台真正的服务器了。

14.3.2　从云主机运行

以前，程序员会为了在计算机上运行或者存储自己的程序而付费。个人电脑发明之后，这

种事情似乎没必要了——人们可以直接在自己的电脑上写程序并运行。现在，应用程序的计算需求已经完全超越了微处理器的发展速度，于是程序员又开始为计算能力付费了。

但是，这次用户不再为单个物理机器的计算能力付费，而是为多个机器共同的计算能力付费。这种云状计算系统的计算能力可以按使用时间进行付费。例如，当客户计算的低成本比即时性更重要时，亚马逊的 EC2 允许用户使用“竞价型实例”（spot instance），可以先竞价再使用云计算服务。

计算实例还可以进行定制，也可以根据应用程序的实际需求进行设置，选项有“高内存”“快速计算”“大容量存储”。虽然网络爬虫不需要很多内存，但是你可能需要较大的存储空间或快速的计算能力来实现爬虫的更多功能。如果你要做大量的自然语言处理、OCR 或者路径查找（就像“维基百科六度分隔理论”问题）之类的工作，选择“快速计算”实例就可以。如果你要采集大量数据，存储许多文件，或者进行大数据分析，可能就需要用带大容量存储的计算实例了。

虽然云计算的花费可能是无底洞，但是写到这里的时候，启动一个计算实例最便宜只要每小时 1.3 美分（亚马逊 EC2 的 micro 实例，其他实例会更贵），Google 最便宜的计算实例是每小时 4.5 美分，最少需要用 10 分钟。考虑计算能力的规模效应，从大公司买一个小型的云计算实例的费用，和自己买一台专业实体机的费用应该差不多——不过用云计算不需要雇人去维护设备。

显然，一步一步设置和运行云计算实例的教程超出了本书介绍范围，不过你自己其实不需要这类教程。亚马逊和 Google（还有不计其数的小公司）的云计算产品正在激烈竞争，它们会尽力把新实例创建的步骤做到最简单，填个应用名称，设置一下信用卡信息就可以了。写到这里的时候，亚马逊和 Google 还为新用户提供了价值几百美元的免费计算时间。

设置好计算实例之后，你就有了新 IP 地址、用户名，以及可以通过 SSH 进行实例连接的公私密钥了。后面要做的每件事情，都应该和你在实体服务器上干的事情一样了——当然，你不需要再担心硬件维护，也不用运行复杂多余的监控工具了。

14.4　其他资源

很多年以前，“在云端”运行基本上是那些既懂理论又具有服务器运维经验的人们之间的高谈阔论。但是今天，由于云计算技术的不断普及，以及云计算供应商之间的竞争，云计算工具已经有了极大的改善。

如果你想建立规模更大或更复杂的爬虫，在创建云计算平台以收集和存储数据时，可能还需要一些参考资料。

Marc Cohen、Kathryn Hurley 和 Paul Newson 合著的 *Google Compute Engine*（http://shop.

oreilly.com/product/0636920028888.do）是通过 Python 和 JavaScript 使用 Google 云计算平台的第一手资料。书中不仅介绍了 Google 的用户界面，还介绍了命令行和脚本工具，可以让你的应用获取更大的灵活性。

如果你更喜欢亚马逊，Mitch Garnaat 的 *Python and AWS Cookbook*（http://shop.oreilly.com/product/0636920020202.do）是一本非常实用的手册，可以让你顺利启动 AWS 服务，还会告诉你如何创建并运行一个可扩展的应用。

14.5 勇往直前

网络一直在不断地变化。那些给我们带来了图像、视频、文字和其他数据文件的计算机技术也在不断地升级和改进。如果想紧跟技术潮流，采集互联网数据的技术就需要随机应变。

本书未来的版本可能会完全忽略 JavaScript，它已是一种过时的、极少用的技术了，而重点关注用 HTML8 实现页面。但是，采集网站内容的基本思路和一般方法是不会改变的。无论现在还是将来，遇到一个网络数据采集项目时，你都应该问问自己以下几个问题。

- 我需要回答或要解决的问题是什么？
- 什么数据可以帮助我，它们都在哪里？
- 网站是如何展示数据的？我能准确地识别网站代码中包含信息的部分吗？
- 该如何定位这些数据并获取它们？
- 为了让数据更实用，我应该做怎样的处理和分析？
- 怎样才能让采集过程更好，更快，更稳定？

总之，你不仅需要掌握如何使用本书中介绍的工具，还要知道如何把它们有效地组合起来解决问题。有时，数据格式很规范，比较容易获取，用一个简单的爬虫就搞定了。有时，你可能需要仔细地思考一番才能解决。

例如，在第 10 章里，我首先用 Selenium 获取在亚马逊图书预览页面中通过 Ajax 加载的图片，然后再用 Tesseract 读取图片，识别里面的文字。在“维基百科六度分隔”问题中，我先用正则表达式实现爬虫，把维基词条链接信息存储到数据库，然后用有向图算法寻找词条凯文・贝肯与词条埃里克· 艾德尔之间最短的链接路径。

在使用自动化技术采集互联网数据时，其实很少遇到完全无法解决的问题。记住一点就行：互联网其实就是一个用户界面不太友好的超级 API。

附录A

Python简介

根据美国计算机协会（Association of Computing Machinery，ACM）的统计数据，Python 是全美国最流行的计算机教学语言，在编程入门课程中它比 BASIC、Java 甚至 C 语言更流行。这部分内容将介绍 Python 3.x 版本和 Python 程序的安装、使用和运行。

安装与“Hello,World!”

如果你用 Mac OS X 或是任意版本的 Linux，Python 可能已经安装在系统上了。如果不确定，可以输入下面的命令检查一下：

```
$python --version
```

我们全书都用 Python 3.x 版本。如果你发现系统安装了 Python 2.x 版本，你可以升级系统，用 `apt-get` 命令安装：

```
$sudo apt-get install python3
```

注意，如果要用 Python 3.x 版本运行程序，你需要在命令行里输入 `$python3 myScript.py`，而不是 `$python myScript.py`。

本书写到这里的时候，Mac OS X 操作系统默认已经安装了 Python 2.7 版本。要安装 Python 3.x 版本，请从 Python 官网下载安装包（https://www.python.org/downloads/mac-osx/）。另外，如果你需要同时运行 Python 2.x 版本和 Python 3.x 版本，那么在使用 Python 3.x 版本时需要使用 `$python3`。

Windows 系统默认没有 Python，在 Python 官网上有预编译安装包（https://docs.python.org/3/using/windows.html）可以下载。只需简单地下载，打开，然后安装。虽然在 Windows 系统上，你可能还需要在系统设置里把 Python 安装路径添加到系统变量中，让操作系统能够找到它，但是网站和安装包都会提供非常简单的操作指导帮你完成这些。

关于各个系统平台中安装和升级 Python 的更全面内容，请访问 Python 下载页面（https://www.python.org/downloads/）。

Python 可以作为脚本语言使用，不需要像 Java 一样创建类和函数。直接打开文本编辑器，写入：

```
print("Hello, Internet!")
```

然后把文件保存为 hello.py 就可以了。如果你想更细致一点儿，可以创建一个函数来实现同样的结果：

```
def hello():
    print("Hello, Internet!")
hello()
```

这里有几点需要注意一下。

Python 不用分号表示句子结束，也不用大括号表示循环体或函数体的开始和结束。Python 用断行（line break）和缩进（Tab 键）来控制代码逻辑。在上面的文件中，第一行是 `def`，后面跟着函数名和参数列表（这里我们用 () 表示没有参数），还有一个分号表示下一行开始就是函数体。函数声明下面的函数体中每一行都必须缩进，结束缩进表示退出函数体。

一开始可能会觉得用空格表示缩进有点儿傻，但是请坚持。以我的经验看，写 Python 会改善我们写其他代码的可读性。不过切换到其他编程语言时，比如 Java 或 C 语言，记得在句末加上分号。

Python 是一个弱类型语言，就是说变量初始化的时候不需要显式声明变量类型（字符串、整数、对象等）。和其他弱类型语言一样，这样有时候调试会出问题，不过这样声明变量很简单：

```
greeting = "Hello, Internet!"
print(greeting)
```

就这些！你已经是一个 Pythoneer 了！

显然，这点儿知识只是 Python 的冰山一角，但是 Python 是一门非常简单的语言，其他语言的程序员甚至可以不用预先了解太多就可以阅读和理解它的代码。这门语言的简单特性，可以用 Python 最著名的“彩蛋”（Easter Egg）来总结：

```
import this
```

输出结果是：

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than right now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

翻译过来即是：

```
《Python之禅》  Tim Peters

优美胜于丑陋
明了胜于隐晦
简洁胜于复杂
复杂胜于混乱
扁平胜于嵌套
宽松胜于紧凑
可读性很重要
即便是特例,也不可违背这些规则
虽然现实往往不那么完美
但是不应该放过任何异常
除非你确定需要如此
如果存在多种可能,不要猜测
肯定有一种——通常也是唯一一种——最佳的解决方案
虽然这并不容易,因为你不是Python之父[1]
动手比不动手要好
但不假思索就动手还不如不做
如果你的方案很难懂,那肯定不是一个好方案
如果你的方案很好懂,那肯定是一个好方案
命名空间非常有用,我们应当多加利用
```

注 1：这一行可能是指荷兰计算机科学家艾兹赫尔·戴克斯特拉（Edsger Dijkstra），他在 1978 年曾经说过："我认为，编程语言设计的一个原则是……无论从哪一点来看，等价的程序不太可能有不一样的表示……但是，完全不同的编程方式增加了不必要性，因此妨碍了代码的维护效率、可读性等。"（http://www.cs.utexas.edu/~EWD/transcriptions/EWD06xx/EWD660.html）不过这句话可能就是指 Python 的发明人，Guido van Rossum，他也是荷兰人。不过好像没人能确定到底是哪一种情况。美国亚利桑那州全州不用夏令时，凤凰城是其州府。——译者注

附录B

互联网简介

互联网是一种信息交换形式，需要处理的内容越来越复杂，相关术语和技术的复杂性也在增加。互联网最早只是作为一种科研信息的交流方式，而现在它还需要处理大文件上传、流媒体视频、安全的银行结算、信用卡交易，以及传递企业间的机密文件。

尽管互联网的多层结构（OSI 模型）非常复杂，但究其本质依然是由一组消息构成的。一些消息用于信息请求，一些消息用于不断地响应请求，还有一些消息包含发给机器的某个应用程序的文件信息或指令。这些请求不断地从一个客户端（桌面或移动设备）发送到一台服务器上，循环往复。它们也会在服务器之间相互传递，有的是用于收集更多客户端的信息。

图 B-1 描绘了几个常见的互联网数据交换方式：请求一个域名的服务器地址，在两个服务器间请求网页和页面图片，还有一个请求实现图片上传。

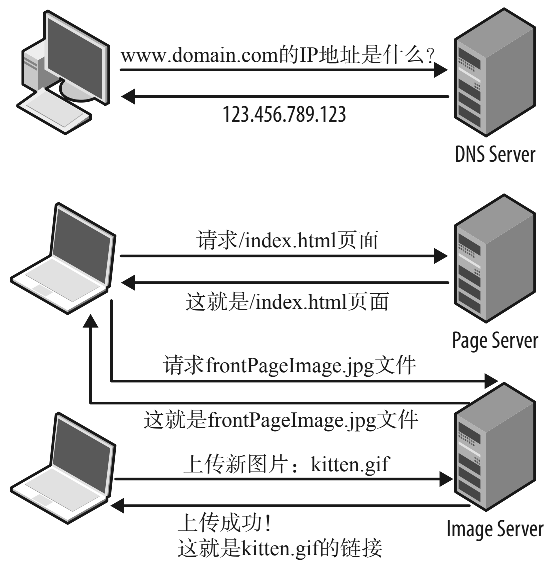


图 B-1：若干常见客户端 – 服务器互联网模型数据交换方式

有许多不同类型的协议或语言控制客户端和服务器之间的这些通信方式。你可以通过 SMTP 协议收邮件，通过 VOIP 协议打电话，还可以通过 FTP 上传邮件。每个协议都为请求头定义了不同的字段，采用不同的数据编码、收发地址或名称，以及其他数据类型。用于网站信息的请求、发送和接收的协议是 HTTP（Hypertext Transfer Protocol，超文本传输协议）。

对本书中的绝大多数爬虫（以及你要编写的大多数爬虫）来说，HTTP 是用来和远程网络服务器通信的。因此，需要对这个协议多做一点儿介绍。

一个 HTTP 消息包括两部分：头字段（header field）和数据字段（data field）。每个头字段由一对标题和值构成。这些字段的标题是 HTTP 标准预先定义好的。比如，你可能会看到一个头字段是：

```
Content-Type: application/json
```

它表示 HTTP 数据包中的数据将用 JSON 格式。在一个 HTTP 数据包里面可能会出现 60 多种头字段，但是在本书里我们只介绍一小部分。下表中的一些 HTTP 头字段应该都是你比较熟悉的：

名　称	描　述	示　例
User-Agent	字符串，表示发出请求的浏览器和操作系统信息	Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:28.0) Gecko/20100101 Firefox/28.0

（续）

名　　称	描　　述	示　　例
Cookie	变量，被网络应用用来存储会话数据和其他信息	“_utma:20549163.147923691.1398729710.1398729710.1398858679.2”
Status	代码，表示网页请求成功与否	“200”（成功），“404”（文件未找到）

当 HTTP 从浏览器收到一个数据包，数据包的内容一定被看成一个网站。网站的结构由 HTML（HyperText Markup Language，超文本标记语言）构成。虽然 HTML 通常被看成编程语言，但它其实是一个标记语言。它通过标签定义文档结构以确定各个元素，像标题（title）、正文（content）、侧边栏（sidebar）、页脚（footer）等。

所有的 HTML 网页（至少是格式正常的页面）都是用 `<html></html>` 标签开始结束，里面还会有 `<head>` 和 `<body>` 标签。其他标签都被放在 `<head>` 和 `<body>` 标签里构成页面的内容：

```
<html>
<head>
<title>An Example Page</title>
</head>
<body>
<h1>An Example Page</h1>
<div class="body">
Some example content is here
</div>
</body>
</html>
```

在这个例子中，页面标题（会显示在浏览器的标签页）是“An Example Page”，同样的标题也放在 `<h1>` 标签里。之后是一个 `class` 属性为 `body` 的 `div` 标签，里面会包括文章或大段的文字。

分析网站以便轻松采集

在网页的不同区域中准确地复制页面内容，对网络数据采集是很有用的——页面上的一部分内容可能比其他部分更容易采集。

在上面的示例页面“An Example Page”中，标签可以通过两个明显的分隔区域获取，但是网络中还会出现更复杂的情况。例如，你可能需要在一个公司网站采集员工信息，而员工名称的格式并不统一，如下所示：

```
<span id="mary_smith">Dr. Mary Smith, CEO</span>
<span id="john_jones">President of Finance Mr. John E. Jones</span>
<span id="stacy_roberts">Stacy Roberts III, Marketing</span>
```

要把这些员工信息抽取成“firstname, lastname”的形式很难。怎么处理带有中间名、职称和其他附加数据的员工信息呢？这里的 `id` 属性的内容格式看着很合适，可以直接用 Python 的字符串分割函数 `split` 对下划线分割来获得结果。

CSS（Cascading Style Sheets，层叠样式表）是配合 HTML 对网站样式进行定义的语言。CSS 可以为网站对象定义颜色、位置、尺寸和背景色等属性。

在上面的示例页面中增加 CSS，如下所示：

```
h1{
    color:'red';
    font-size:1.5em;
},
div.body{
    border:2px solid;
}
```

这段代码会在网站中实现一个适当尺寸的红色标题，并在正文内容外增加一个边框。

不过，HTTP、HTML、CSS 的内容非常丰富，本书不做更多介绍。如果你对这些内容很陌生，我推荐你到 W3Schools（http://www.w3schools.com/）去查询本书中出现的 HTML/CSS 术语和代码。利用浏览器的“查看源代码”功能，也可以很快地熟悉这些内容。

附录C

网络数据采集的法律与道德约束

2010 年，软件工程师 Pete Warden 做了一个网络爬虫，从 Facebook 上收集数据。他一共收集了大约两亿个 Facebook 用户的用户名、位置、好友和兴趣爱好信息。当然，Facebook 发现了这个行为，并给他发了一个要求“停止并终止”的邮件，他遵守了。有人问他为什么要遵守 Facebook 的要求，他说：“大数据虽然很便宜，但律师费可不便宜。”

在这一章里，我们将介绍美国及其他国家现存的一些与网络数据采集相关的法律，并学习如何分析网络数据采集行为的法律和道德约束。

在阅读下面的内容之前，希望你能理解：我是软件工程师，不是律师。不要把在本章或本书其他章节学到的相关法律知识看成专业的法律意见或规范。虽然我相信自己有足够的能力，可以介绍网络数据采集行为的法律和道德约束，但是在做那些可能要承担法律责任的网络数据采集项目之前，你还是应该咨询一下律师，而不是软件工程师。

C.1　商标、版权、专利

现在，我们开始知识产权（Intellectual Property）第一课！知识产权有三种基本类型：商标（用 ™ 或 ® 表示），版权（用 © 表示），专利（有时会在文字里出现专利保护相关说明，但通常没有任何说明）。

专利用来声明内容所有权仅属于发明者。你不能为任何图片、文字和信息本身取得专利。虽然有些专利，比如软件专利，并不像我们通常理解的“发明创造”那样是有形的，但是要注意这些无形的东西（技术）是有专利权的——并不是专利报告中的内容。除非你用采集来的设计图来盖楼，或者使用了一种具有专利权的网络数据采集方法，否则你不太可能

在网络数据采集时侵犯他人的专利。

虽然商标也不太可能出现问题，但是有些事情还是需要注意的。美国专利商标局对商标的介绍如下：

> **商标**（trademark）是一个单词、词组、符号和 / 或设计，用来识别和区分一种商品的来源。**服务标识**（service mark）是一个单词、词组、符号和 / 或设计，用来识别和区分一种服务而非商品的来源。术语“商标”通常既可表示商标，也可表示服务标识。

除了当我们提到商标时通常会想到的传统的单词 / 词组商标，其他的描述性特征也可以作为商标。比如，容器的外形（可口可乐的瓶子），或者一种颜色（美国欧文斯科宁的 Pink Panther 玻璃纤维隔热层的粉色）。

和专利不同，商标的所有权很大程度上由使用场景决定。比如，如果我想在博客里发一篇带可口可乐图标的文章，那么做是没问题的（只要我没有暗示我的博文是可口可乐赞助或发布的就行）。但是，如果我想做一种新的碳酸饮料，在外包装也用可口可乐图标，那明显就侵权了。同样道理，虽然我可以把饮料外包装涂成 Pink Panther 的粉色，但是我不能用同样的颜色发行一种新的家用隔热层产品。

版权法

商标和专利通常都要正式地登记，以便人们知晓。与一般认识不同的是，具有版权的物品并不都很容易识别。究竟带什么因素的图像、文字、音乐有版权呢？并不是说在网页下面加上“保留所有权利”（All Rights Reserved）就具有版权，也不是“公开出版发行的”（published）与“未公开出版发行的”（unpublished）的物品版权就不一样。只要你把一件东西带到世间，它就会自动受到版权法的保护。

《保护文学和艺术作品伯尔尼公约》是 1886 年由瑞士政府在伯尔尼首次公布的版权国际标准。这个公约的基本含义是所有成员国都必须像对待自己国家的公民的作品一样，对其他国家公民的作品进行版权保护。其实，就是说作为一个美国公民，如果你涉嫌抄袭一个法国公民的作品，也要承担法律责任（反之亦然）。

显然，版权是网络爬虫需要关注的内容。如果我采集别人的博客然后放到我自己博客上，我就可能会惹上官司。不过，我有几层保护，可以根据博客采集项目的实际影响，帮我进行辩护。

首先，版权保护只涉及创造性的作品。它不会涉及统计数据或事实。好在许多网络爬虫采集的都是事实和统计数据。虽然一个网络爬虫从网络上收集诗歌，然后显示在自己的网站上有可能是违反版权法的，但是如果它收集不同时间段诗歌发表的数量就不违法了。诗歌是一种创造性作品，但是按月对网站上的诗歌进行字数统计，就没什么创造性了。

如果数据是公司发布的价格、高管的姓名或者其他事实性的信息，那么即使完全照搬（不是根据采集的原始数据进行整合或计算）也不会违反版权法。

按照《数字千年版权法》（Digital Millennium Copyright Act，DMCA），即使是有版权的内容也可以以合理理由直接使用。DMCA 列举了一些对有版权的内容进行自动收集的规则。DMCA 非常长，包含了从电子书到电话的许多细则。但是，有三点需要格外注意。

- 根据"安全遮蔽"（safe harbor）保护原则，如果你从一个你认为是无版权的数据源中采集数据，但是有人曾向那个数据源申请过版权，那么只要你在得到提醒后把有版权的材料删除，就可以免责。
- 为了收集信息，你不能用手段故意绕开安全措施（比如密码保护）。
- 你可以根据"公平使用"（fair use）原则使用信息，但需要考虑有版权作品占总信息的百分比，以及使用这类有版权作品的目的。

总之，未经作者或版权所有者的授权，你不可以直接发布有版权的信息。如果你以数据分析为目的，把允许自由使用的有版权的信息保存在自己的不公开数据库里，是合法行为。如果你把数据展示到网站上供人们浏览或下载，就不算合法了。如果你分析数据库里的数据，发布作品的字数统计，按作品数量排列的作者，或其他的数据分析结果，是合法行为。如果你还引用了一些原文，或简单的样本数据来阐述自己的观点，也是可以的，但是使用之前最好看看 DMCA 里"公平使用"原则的条例。

C.2　侵犯动产

侵犯动产与我们常识中的"违法"有着本质的区别，动产的范围不包括不动产和土地，是指那些可移动的财产（比如服务器）。如果接入那些不允许你接入或使用的财产，就会侵犯动产。

在目前的云计算时代，人们可能不把网络服务器看作一种真实有形的资源。但其实服务器不仅由许多昂贵的组件构成，而且它们还需要空间存放、监控、制冷，以及大量的电力供应。据估计，全球 10% 的电力都是由计算机消耗的（如果你自己的电费构成并非如此，可以考虑一下 Google 庞大的服务器农场，每一座都需要与大型电站连接）。

虽然服务器是很昂贵的资源，但是从法律角度看，一种非常有趣的现象是，网站运营者非常希望人们消费他们的资源（接入他们的网站），但同时他们又不希望资源被过快得消耗掉。通过浏览器看一下网站可以，但是用大规模的 DDOS 攻击就不允许了。

如果满足下列三个条件，网络爬虫就属于侵犯动产。

- *缺少许可*

  由于网络服务器对每个人都是开放的，所以它们一般也会向网络爬虫"提供许可"。但

是，很多网站的服务协议条款都明确地禁止使用爬虫。另外，任何要求终止邮件中也会明确地废除这类许可。

- 造成实际的伤害
  服务器是很昂贵的。除了服务器成本，如果你的爬虫把网站拖垮了，或者限制了网站为其他用户提供服务的能力，这些都算是你对网站造成的“伤害”。

- 故意而为
  这个，你懂的！

只有三个条件都满足才算是侵犯动产。然而，如果你违反了服务协议，并没有造成实际伤害，不要认为你就不算违法。可能你的行为也已经违法了版权法、DMCA、《计算机欺诈与滥用法》（The Computer Fraud and Abuse Act，CFAA，后面会详细介绍），或者其他可以处理网络爬虫犯罪行为的法律。

请限制你的爬虫

过去，网络服务器比个人电脑要强大得多。其实，“服务器”的部分定义就是指“大型计算机”。而现在情况似乎已经倒过来了。比如，我自己的笔记本，3.5 GHz 处理器，8G 内存。一个亚马逊的中等云计算实例（写到这里的时候）却只有 3 GHz 处理器和 4G 内存。

如果网速正常，还有一台可以持续采集的专用设备，即使一台个人电脑也可以给许多网站造成沉重负担，甚至可以对网站造成严重伤害或直接把网站拖垮。除非出现了紧急医疗事故，而唯一的援救方法是在两秒内收集《阿周真人秀》（Joe Schmo）网站上所有的搞笑视频，否则真的没有理由去伤害别人的网站。

一直被盯着看的机器人是永远不会完成的（采集总是需要很长时间）。有时候最好还是让爬虫在午夜运行，而不是在下午或者傍晚运行，原因如下。

- 如果你有大约八个小时的时间，即使采集一页需要的时间是 2 秒，那么你也可以抓 14 000 多页。当时间不怎么紧张的时候，没必要加快爬虫的采集速度。
- 假如网站的目标访客和你在同一时区（如果不在同一时区可以自动调整时间），那么晚上网站流量可能会少很多，这样你的采集行为就不会影响网站高峰期的运行了。
- 你可以用爬虫采集的时间睡觉，不用为了看到新信息而不断地翻日志。想想看，第二天早上睡醒的时候崭新的数据就摆在面前，得有多么惬意啊！

再想象一下下面三种场景。

- 你有一个网络爬虫遍历了《阿周真人秀》网站，收集了一些或全部的数据。
- 你有一个网络爬虫遍历了几百个小网站，收集了一些或全部的数据。
- 你有一个网络爬虫遍历了一个超大型网站，比如维基百科。

在第一个场景中，最好让爬虫在深夜慢慢地运行。

在第二个场景中，最好用循环制快速地采集每个网站，而不是一次一个慢慢地采集。根据你要采集的网站数量进行合理安排，这样做可以让你尽可能以最快的网络连接和最多的机器收集数据，而且对每个网站服务器的负载也比较合理。你可以用很多程序实现这种循环采集方式，可以用多线程（每个线程单独采集一个网站，可以暂停），也可以用 Python 队列来跟踪网站。

在第三个场景中，可能你的网络连接和个人电脑对维基百科这样的超大型网站造成的负载不会引起对方的注意。但是，如果你用分布式网络设备采集，显然就不是一回事儿了。请谨慎使用，最好问问对方允不允许这么做。

C.3 计算机欺诈与滥用法

在 20 世纪 80 年代早期，计算机从学术领域走向商业世界。病毒和蠕虫不再仅仅被认为是麻烦事（或者一种业余爱好），而是可能导致实际财务损失的严重犯罪事件。为此，美国联邦政府在 1986 年出台了《计算机欺诈与滥用法》。

尽管你可能会认为这个法律只是针对那些发布病毒的恶意黑客，但其实它对网络爬虫也有很大的影响。假如一个爬虫用简单易猜的密码提交登录表单，对网站进行暴力破解，或者收集偶然置于隐蔽但公开位置的政府机密。这些行为在 CFAA 中都是非法的。

这个法律定义了七种主要犯罪行为，总结如下。

- 明知没有授权，却进入美国政府的计算机，并获取信息。
- 明知没有授权，却进入计算机，并获取财务信息。
- 明知没有授权，却进入美国政府的计算机，影响政府计算机的使用。
- 为了诈骗的目的故意地进入任何受保护的计算机。
- 没有授权的情况下故意地进入一台计算机并导致计算机损坏。
- 分享或销售美国政府使用的计算机，或者影响洲际或国际商务往来的计算机的密码或授权信息。
- 试图通过破坏或威胁破坏任何受保护的计算机，诈骗钱财或“其他利益”。

总之，远离那些受保护的计算机，不要接入没有授权的计算机（或网站服务器），尤其是避开政府或财务计算机。

C.4 robots.txt和服务协议

从法理上说，网站的服务协议和 robots.txt 是很有趣的。如果一个网站允许公众接入，那么网站管理员对软件可以接入什么和不可以接入什么的限制是不合理的。如果网站管理员对

你说，“你用浏览器访问网站没问题，但是你用自己写的程序访问它就不行”，这就不太靠谱了。

大多数网站在每页页脚都有自己的服务协议。TOS 不仅包含网络爬虫和自动接入的规则，而且还包括网站收集的信息类型和信息用途，通常还有一条不承担责任的法律声明，提示用户网站提供的服务没有任何费用也不做任何保证。

如果你了解搜索引擎优化（Search Engine Optimization，SEO）或搜索引擎技术，那么你可能听说过 robots.txt 文件。如果你想在任何大型网站上查找 robots.txt 文件，可以在网站根目录 http://website.com/robots.txt 找到。

rebots.txt 文件是在 1994 年出现的，那时搜索引擎技术刚刚兴起。从整个互联网寻找资源的搜索引擎，像 AltaVista 和 DogPile，开始和那些把网站按照主题进行分类的门户网站公司激烈竞争，比如像 Yahoo! 这样的门户网站。互联网搜索规模的增长不仅说明网络爬虫数量的增长，而且也体现了网络爬虫收集信息的能力在不断进化。

虽然我们今天认为这种能力是十分平常的，但是当自己网站文件结构深处隐藏的信息变成了搜索引擎首页上可以检索的内容时，有些网站管理员还是会感到震惊。于是，rebots.txt 文件，也称为机器人排除标准（Robots Exclusion Standard），应运而生。

与通常用人类语言宽泛地讨论网络爬虫的 TOS 不同，rebots.txt 文件可以被程序轻易地解析和使用。虽然它看着好像可以完美地解决所有爬虫的问题，毕其功于一役，但是请注意下面两种情况。

- rebots.txt 文件的语法没有标准格式。它是一种业内惯用的做法，但是没有人可以阻止别人创建自己版本的 rebots.txt 文件（并不是说如果它不符合主流标准，机器人就可以不遵守）。它是一种被企业广泛认可的习惯，主要是因为这么做很直接，而且企业也没有动力去发展自己的版本，或者尝试去改进它。
- rebots.txt 文件并不是一个强制性约束。它只是说“请不要抓网站的这些内容”。有很多网络爬虫库都支持 rebots.txt 文件（虽然这些默认设置很容易修改）。另外，按照 rebots.txt 文件采集信息比直接采集要麻烦得多（毕竟，你需要采集、分析，并在代码逻辑中处理页面内容）。

机器人排除标准的语法很直接。和 Python 等语言一样，注释都是用 `#` 号，用换行结尾，可以用在文件的任意位置。

文件的第一行非注释内容是 `User-agent:`，注明具体哪些机器人需要遵守规则。后面是一组规则 `Allow:` 或 `Disallow:`，决定是否允许机器人访问网站的该部分内容。星号（`*`）是通配符，可以用于 `User-agent:`，也可以用于 URL 链接中。

如果一条规则后面跟着一个与之矛盾的规则，则按后一条规则执行。例如：

```
#Welcome to my robots.txt file!
User-agent: *
Disallow: *

User-agent: Googlebot
Allow: *
Disallow: /private
```

在这个例子中，所有的机器人都被禁止访问网站的任意内容，除了 Google 的网络机器人，它被允许访问网站上除了 /private 位置的所有内容。

Twitter 的 robots.txt 文件对 Google、Yahoo!、Yandex（俄罗斯著名搜索引擎）、微软，以及其他机器人或搜索引擎访问范围都有明确的要求。Google 搜索（和其他机器人的访问范围一样）的内容如下所示：

```
#Google Search Engine Robot
User-agent: Googlebot
Allow: /?_escaped_fragment_

Allow: /?lang=
Allow: /hashtag/*?src=
Allow: /search?q=%23
Disallow: /search/realtime
Disallow: /search/users
Disallow: /search/*/grid

Disallow: /*?
Disallow: /*/followers
Disallow: /*/following
```

Twitter 之所以采取如此严格的访问限制是因为它提供了 API。因为 Twitter 的 API 更容易控制（可以通过授权赚到钱），所以 Twitter 会禁止任何使用爬虫采集网页来收集网站信息的机器人。

虽然看到一个指明爬虫采集限制范围的文件让人感觉很憋屈，但是它其实可以成为网络爬虫开发的指示灯。如果你发现一个 robots.txt 文件禁止采集网站上某个部分的内容，那么基本可以确定网管同意你采集其他部分的所有内容（如果他们不愿意让你采集，在 robots.txt 文件中应该已经明确禁止了）。

例如，维基百科的 robots.txt 文件规定了网络爬虫（并非搜索引擎）非常好的权限。甚至只要是人类可以阅读的文字都运行机器人采集（适合我们的爬虫！），只会禁止一小部分页面，比如登录页面、搜索页面和“随机词条”页面：

```
#
# Friendly, low-speed bots are welcome viewing article pages, but not
# dynamically generated pages please.
#
# Inktomi's "Slurp" can read a minimum delay between hits; if your bot supports
# such a thing using the 'Crawl-delay' or another instruction, please let us
```

```
# know.
#
# There is a special exception for API mobileview to allow dynamic mobile web &
# app views to load section content.
# These views aren't HTTP-cached but use parser cache aggressively and don't
# expose special: pages etc.
#
User-agent: *
Allow: /w/api.php?action=mobileview&
Disallow: /w/
Disallow: /trap/
Disallow: /wiki/Especial:Search
Disallow: /wiki/Especial%3ASearch
Disallow: /wiki/Special:Collection
Disallow: /wiki/Spezial:Sammlung
Disallow: /wiki/Special:Random
Disallow: /wiki/Special%3ARandom
Disallow: /wiki/Special:Search
Disallow: /wiki/Special%3ASearch
Disallow: /wiki/Spesial:Search
Disallow: /wiki/Spesial%3ASearch
Disallow: /wiki/Spezial:Search
Disallow: /wiki/Spezial%3ASearch
Disallow: /wiki/Specjalna:Search
Disallow: /wiki/Specjalna%3ASearch
Disallow: /wiki/Speciaal:Search
Disallow: /wiki/Speciaal%3ASearch
Disallow: /wiki/Speciaal:Random
Disallow: /wiki/Speciaal%3ARandom
Disallow: /wiki/Speciel:Search
Disallow: /wiki/Speciel%3ASearch
Disallow: /wiki/Speciale:Search
Disallow: /wiki/Speciale%3ASearch
Disallow: /wiki/Istimewa:Search
Disallow: /wiki/Istimewa%3ASearch
Disallow: /wiki/Toiminnot:Search
Disallow: /wiki/Toiminnot%3ASearc
```

是否按照 robots.txt 文件的要求写网络爬虫是由你自己决定的，当爬虫毫无节制地采集网站的时候，强烈建议你遵守。

C.5　三个网络爬虫

因为网络数据采集是一个没有限制的领域，所以有很多意想不到的情况会让你置身于恢恢法网之中。在这一节里，我们将介绍三个网络爬虫相关的法律案件，看看案件里网络爬虫是如何被使用的。

C.5.1　eBay起诉Bidder’s Edge与侵犯动产

1997 年，豆宝宝（Beanie Baby）市场依旧如火如荼，科技领域的泡沫不断膨胀，在线房屋

拍卖也已经成为互联网的新热点。有一家叫 Bidder’s Edge 的公司创造了一种新的线上拍卖模式。客户不需要从一个拍卖网站到另一个拍卖网站查看商品价格，这个公司可以把所有网站同一商品（比如一个流行的 Furby doll 玩具或 Spice World 电影）的信息放到一起进行比价，然后你就可以很方便地点击最低价的网站去购买了。

Bidder’s Edge 通过很多网络爬虫实现了这个网站，为了得到商品价格和信息，不断地向各大拍卖网站的网络服务器发起请求。在当时的拍卖网站中，最大的是 eBay，Bidder’s Edge 每天要向 eBay 服务器请求 100 000 次。就算按照今天的观点看，这也是很大的流量。据 eBay 公布的数据显示，这相当于网站一天总流量的 1.53%，老板肯定不会开心了。

eBay 给 Bidder’s Edge 发了一封要求终止的警告邮件，以及一个 eBay 数据授权申请表。但是，授权谈判没成功，Bidder’s Edge 仍然一意孤行，继续爬取 eBay 的数据。

虽然 eBay 封杀了 Bidder’s Edge 的 169 个 IP 地址，但是 Bidder’s Edge 还是通过代理服务器继续采集（发送请求的时候显示代理服务器的 IP）。“军备竞赛”就这样开始了，双方僵持不下——Bidder’s Edge 在被封杀之后不断启用新代理服务器和新 IP 地址，eBay 则不断更新防火墙列表（增加对每个可疑 IP 地址发送的数据包进行检查）。

最终，在 1999 年 12 月，eBay 起诉 Bidder’s Edge 侵犯动产。（https://en.wikipedia.org/wiki/EBay_v._Bidder%27s_Edge）

因为 eBay 的服务器是 eBay 拥有的真实有形的资源，它不想让 Bidder’s Edge 滥用自己的资源，使用侵犯动产起诉好像非常合理。实际上，目前侵犯动产在网络爬虫法律案件中十分普遍，也经常被 IT 法律案件使用。

法院当时规定 eBay 出示两方面证据才可以证明自己被侵犯动产：

- Bidder’s Edge 未经允许使用 eBay 资源
- eBay 确实因为 Bidder’s Edge 的行为遭受了经济损失

由于之前 eBay 发过要求终止警告信，而且 IT 日志可以显示服务器使用情况以及服务器相关的财务成本，所以 eBay 很容易就提供了证据。当然，大型法律案件都不会轻松结束：双方来回扯皮，支付了大量律师费，最终在 2001 年 3 月法院才最终判决。

那么，这个案例是不是说，以后只要任何人未经授权使用别人的服务器，就是侵犯动产了呢？也不一定。Bidder’s Edge 是一个极端案例：它用了 eBay 太多的资源，导致 eBay 不得不购买更多的服务器，花更多电费，可能还要多雇人维护（虽然 1.53% 看着并不多，但对这样的大公司来说所有加总肯定是一笔大数目）。

2003 年，加州高级法院判了另一个案子，Intel 公司起诉 Hamidi 失败。Intel 前雇员 Hamidi 通过 Intel 服务器向 Intel 的员工发送让 Intel 不爽的邮件。法院结案时说：

> Intel 败诉并不是因为通过网络发送邮件可以不承担任何法律责任，而且因为侵犯动产的民事侵权行为不成立——并不像这个案例中描述的——如果原告没有证据证明自己财产或法律权益受到损失，在加州就不能胜诉。

最后，Intel 无法向法院证明 Hamidi 向公司其他员工发送的六封邮件给公司造成了经济损失（有趣的是，每封邮件都带有已经从 Hamidi 邮箱删除的选项——说明他还是挺懂规矩的）。这件事并没有给 Intel 造成任何财产损失。

C.5.2　美国政府起诉Auernheimer与《计算机欺诈与滥用法》

如果网上的信息可以让人用浏览器轻而易举地获得，那么你用自动化手段获取同样的信息不太可能会引起联邦调查局调查你。但是，如果一个非常细心的人在网站上发现了一个极小的安全漏洞，再使用网络爬虫自动化采集网站，那么这个极小的安全漏洞就会变得越来越大并且非常危险，被联邦调查局调查就很正常了。

2010 年，Andrew Auernheimer 和 Daniel Spitler 在 iPad 上发现了一个新功能，当用 iPad 访问 AT&T 网站的时候，AT&T 跳转到一个带有 iPad 的唯一 ID 号的链接：

> https://dcp2.att.com/OEPClient/openPage?ICCID=<idNumber>&IMEI=

这个页面包括一个登录表单，上面显示对应 ID 号的用户的邮箱地址，用户只要输入密码就可以访问他们的账号了。

虽然有大量可能的 ID 号，但只要有足够的爬虫，用一串随机数迭代，就可以收集邮箱地址。通过 AT&T 网站的这个登录功能基本上就把用户的邮箱地址公布到网络上了。

Auernheimer 和 Spitler 做了一个爬虫，一共收集了 114 000 个邮箱地址，里面包含知名人士、企业 CEO 和政府官员的邮箱地址。Auernheimer 向高客传媒（Gawker Media）发布了地址列表以及如何获取的方法，高客传媒也很给力，在自己网站发布了头条消息“苹果最烂安全事故：114 000 个用户信息被曝”（不过没有公布邮箱列表）。

2011 年 6 月，Auernheimer 的家突然遭到 FBI 搜查，FBI 索要邮箱地址，不过最终以贩毒罪逮捕了他。2012 年 11 月，他因未经授权接入计算机被判欺诈与共谋罪，之后在联邦监狱关押 41 个月，并要求赔偿 73 000 美元。（https://en.wikipedia.org/wiki/Weev）

他的案子引起了民权律师 Orin Kerr 的关注，他组建律师团队，把案子上诉到了美国联邦第三巡回上诉法院。2014 年 4 月 11 日（这类法律程序时间都比较长），第三巡回上诉法院接受上诉，法院的意见是：

> Auernheimer 在第一法院的罪名必须撤销，因为根据《计算机欺诈与滥用法》，18 U.S.C. § 1030(a)(2)(C)，访问公共网站并不需要经过许可。AT&T 并没有使用密码或任何其他保护措施控制其他人获取用户的邮箱地址。AT&T 主观地希望外人不会偶然看到敏感数据，以及 Auernheimer 的行为被假设成"小偷"都是不恰当的。公司配置服务器使得信息向每个人公开，就是授权一般公众可以查看信息。通过 AT&T 的公共网站获取邮箱地址是 CFAA 允许的行为，因此 Auernheimer 无罪。

于是，理性在法律体系中又一次获得了最终胜利。同一天，Auernheimer 从监狱释放，从此每个人都可以快乐地生活了。

虽然 Auernheimer 被认定没有违反《计算机欺诈与滥用法》，但是他曾经在家中被 FBI 强行搜查，花费了数千美元的律师费，也花了三年时间诉讼和坐牢。作为网络爬虫从业者，我们能从中汲取什么教训，让我们远离这类情况，避免类似情况发生在自己身上呢？

首先，采集任何敏感信息的时候，无论是个人隐私（本案例中是邮箱地址）、交易机密或是政府机密，在向律师咨询之前，都不应该行动。即使信息是公开的，你也要想想："如果普通用户想看这些信息，可以这么容易地得到吗？""这些信息是公司想让用户看的吗？"

其次，我曾经多次给一些公司打电话，告诉他们网站和网络应用的安全隐患。这么说最合适："你好，我是做网络安全的，在你的网站上发现了一个潜在的安全隐患，可以把电话转接到可以处理问题的人那里吗？"对方除了对你的（白帽）黑客天才感激万分，还可能会让你免费订阅网站内容，甚至还会有现金奖励或其他好处！

另外，Auernheimer 向高客传媒发布信息（在通知 AT&T 之前），以及秀自己发现安全隐患的过程，无疑成为了 AT&T 律师的攻击点。

如果你发现了网站的安全隐患，最好的做法就是告诉网管，而不是媒体。尤其是当网站补丁没有及时放出的时候，你可能想写一篇博文然后向世界宣布。但是，你应该记住，那是网站自己的公司该做的事情，与你无关。你最该做的事情就是让你的网络爬虫（也可能是你的生意）远离这些网站！

C.5.3　Field起诉Google：版权和robots.txt

Blake Field 是一名律师，他起诉 Google 违反了版权法，因为当他把自己的书从他的网站上删除之后，Google 还是在搜索引擎里显示了书的副本。版权法允许具有原创性作品的作者控制作品的发布渠道。Field 认为 Google 的缓存（当他把自己的书从他的网站上删除之后）侵犯了他控制作品发布渠道的权利。

Google 网络缓存

Google 网络爬虫（也叫“谷歌机器人”）采集网站的时候，它们会为网站留一个副本，然后放在互联网上。任何人都可以接入这些缓存，用 URL 链接就可以：

http://webcache.googleusercontent.com/search?q=cache:http://pythonscraping.com/

如果你搜索或采集的网站没有了，你可以用这个方法看看副本还在不在。

知道 Google 的缓存功能却没有采取安全措施，并不能帮助 Field 胜诉。其实，他可以在网站直接增加 robots.txt 文件来禁止 Google 机器人缓存他的网站，里面注明哪些页面可以采集，哪些页面不能采集就行。

更重要的是，法院认为根据 DMCA 的安全港（Safe Harbor）条例，Google 可以合法地缓存和显示 Field 的网站：“服务提供商间接或临时把材料存储在由其控制或操作的系统或网络上，不应当作出经济赔偿……不应当承受侵犯版权的责任。”

作者简介

Ryan Mitchell 是一名软件工程师，目前在美国波士顿的 LinkeDrive 公司工作，主要负责开发公司 API 和数据分析工具。Ryan 本科毕业于美国欧林工程学院，目前在哈佛大学继续教育学院攻读硕士学位。在加入 LinkeDrive 公司之前，她在 Abine 公司构建网络爬虫和网络机器人。她经常从事网络数据采集项目的咨询工作，主要面向金融和零售领域。

封面介绍

本书封面上的动物是一只南非穿山甲。穿山甲是一种独居、喜欢夜间活动的哺乳动物，与犰狳、树懒、食蚁兽是近亲。它们主要分布于非洲的东部和南部。非洲还有三种穿山甲，均属濒临灭绝物种。

成年的穿山甲体长 12~39 英寸，体重可达 3.5~73 磅。它们和犰狳类似，身上有深色、浅棕色或橄榄色的鳞甲。幼年穿山甲的身上主要是粉红色的鳞甲。受到威胁时，尾部的鳞甲更像攻击性武器，可以砍伤攻击者。穿山甲还有一种与臭鼬类似的防御策略，可以从肛门附近的腺体中释放出一种酸性恶臭气体。这么做不仅是向攻击者发出警告，还可以标记自己的势力范围。穿山甲的肚子上并没有鳞甲，不过有一点儿毛。

和它们的近亲食蚁兽一样，穿山甲主要以蚂蚁和白蚁为食。它们异乎寻常的长舌头可以在树洞和蚂蚁窝中寻觅食物。它们的舌头比身体还长，可以在不用的时候缩回胸腔里。

虽然穿山甲是独居动物，但是长大以后会居住在很深的地洞里。但是它们经常“霸占”土豚和疣猪弃用的巢穴。不过通过前肢上三个又长又弯的爪子，穿山甲在需要的时候为自己挖一个地洞也不成问题。

O’Reilly 书籍封面上的许多动物都快要绝种了，它们对这个世界都非常重要。如果你想了解帮助它们的相关信息，请参考 animals.oreilly.com。

封面图片取自 Lydekker 的 *The Royal Natural History*。

O'REILLY®

Python网络数据采集

网络上的数据量越来越大，单靠浏览网页获取信息越来越困难，如何有效地提取并利用信息已成为一个巨大的挑战。本书采用简洁强大的Python语言，全面介绍网络数据采集技术，教你从不同形式的网络资源中自由地获取数据。你将学会如何使用Python脚本和网络API一次性采集并处理成千上万个网页上的数据。

本书适合熟悉Python的程序员、安全专业人士、网络管理员阅读。书中不仅介绍了网络数据采集的基本原理，还深入探讨了更高级的主题，比如分析原始数据、用网络爬虫测试网站等。此外，书中还提供了详细的代码示例，以帮助你更好地理解书中的内容。

通过阅读本书，你将能够：

- **解析复杂的HTML页面**
- **遍历多个网页和网站**
- **了解API的基本概念和工作原理**
- **学习存储数据的方法**
- **下载、读取并抽取网络文档中的数据**
- **使用工具和方法清洗格式异常的数据**
- **读取并处理自然语言**
- **越过表单和登录窗口采集数据**
- **学习采集JavaScript的方法**
- **学习图像处理和文字识别方法**

“这本书中的工具和示例帮我轻松地将一些重复性工作自动化了，我可以将省下来的时间用于解决更有意思的问题。这是一本实用手册，非常适合用来解决实际工作中的问题。”

——Eric VanWyk
美国欧林工程学院
电子计算机工程师

Ryan Mitchell是数据科学家、软件工程师，目前在波士顿LinkeDrive公司负责开发公司的API和数据分析工具。此前，曾在Abine公司构建网络爬虫和网络机器人。她经常做网络数据采集项目的咨询工作，主要面向金融和零售业。另著有*Instant Web Scraping with Java*。

PYTHON

封面设计：Karen Montgomery　张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　计算机/数据分析

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-41629-2

ISBN 978-7-115-41629-2
定价：59.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks